FEASIBILITY
OF
DISTRIBUTED MANUFACTURING OPERATIONS CONTROL

By

SERDAR KIRLI

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA
1999

ACKNOWLEDGMENTS

I would like to thank Dr. Sencer Yeralan for his support, encouragement and friendship. This study would not be possible without his vision and guidance. He has been not only an advisor but also a dear friend.

I would like to extend my appreciations to my committee members Dr. Suleyman Tufekci, Dr. John F. Mahoney and Dr. John K. Schueller for their suggestions and comments throughout the study.

TABLE OF CONTENTS

Abstract of Dissertation Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

FEASIBILITY
OF
DISTRIBUTED MANUFACTURING OPERATIONS CONTROL

By

Serdar Kirli

August 1999

Chairman: Dr. Sencer Yeralan
Major Department: Industrial and Systems Engineering

The last two decades have witnessed the emergence of new technologies in computer control and inter-computer communications. More recently, embedded controllers have redefined industrial automation through a new generation of "smart" machine tools (i.e., tools with local data processing capabilities).

New engineering technologies most often lead to new engineering approaches and, in turn, to new engineering practices. The focus of this dissertation is on the implications of these emerging technologies on manufacturing operations control.

Generally speaking, current industrial engineering practices continue to rely heavily on traditional methods for production control. These methods may be characterized by two common points:

a unicentric view of the physical plant, and the treatment of the system components as passive elements, that is, uninvolved in the decision making process.

This study is a trial for a new paradigm, oriented more towards the discovery and evaluation of previously untested ideas. Specifically, it investigates the feasibility of distributed production line operations control where the information processing is locally carried out by networked (communicating) embedded station controllers. The decision making process relies on rule-based local learning from real-time process data. In accordance with the orientation of the study, the decision making process is kept intentionally simple to retain the focus on paradigm development rather than on seeking streamlined policies. Particularly, a minimal set of rules is selected to characterize the behavior of stations and a straightforward learning mechanism is chosen. The work includes a physical implementation and a computer simulation, whose findings are compared. The controller area network (CAN) is selected as the cost-effective communication protocol. A production line comprised of four stations is used as a testbed for the purposes of this study.

Evidence is revealed implying that even with simple rule-based learning, a distributed embedded approach achieves production line control performances comparable to the traditional techniques. Moreover, it is demonstrated that such a control system may be implemented using low-cost embedded

control. The study discovers empirical aspects of distributed production control that present profound justification for future research. Perhaps most importantly, the distributed embedded approach shows promise in cases where system scalability requirements and environmental dynamics are more demanding.

CHAPTER 1
INTRODUCTION

1.1 Background

The control of manufacturing systems has been a fundamental field of work in industrial engineering [15,41]. In the last few decades many approaches have been developed to determine the optimum operating parameters of manufacturing systems [23,75]. Reviewing the pertinent industrial engineering literature, one observes a gradual transition from more qualitative approaches to manufacturing operations control (MOC) toward more quantitative and analytical approaches [67].

Recent quantitative approaches to MOC have lead to optimization models that are most often solved on a computer in an algorithmic fashion. Paralleling the improvements in data processing technologies, the scale and scope of such models have been increasing. However, there may be limits to such improvements. Although it is possible with today's computers to develop models for large-scale manufacturing systems with hundreds of elements, as a whole, industry tends to use the more simplistic techniques, such as push, pull (kanban), constant work-in-process (CONWIP), and so on. This stems partially from the difficulties in modeling and in verifying the analytical solutions, and the simplifying assumptions inherent in most

analytical models, such as linearity, time-homogeneity, and deterministic operating disciplines. However, the use of such simplistic approaches does not always allow the manufacturing system to fulfill its potential in efficiency and effectiveness.

Presently, production control mainly relies on non-adaptive control strategies such as material requirements planning (MRP) [34], kanban [86], and CONWIP [71]. These have all been developed and have become popular over the last few decades. MRP is a push system where raw materials are pushed into the production line periodically based on demand. Once in the system parts flow toward the output without any control of the downstream stations possibly causing inventory build-ups in front of bottleneck stations. Kanban and CONWIP on the other hand are pull systems, which are geared toward reducing the work-in-process (WIP) inventory. Contrary to the push systems, in pull systems the flow of parts is controlled by the downstream stations. CONWIP is based on the principle that the total inventory in the system remains constant at all times. Hence, a new part can enter the system only after a finished part is removed from the finished goods inventory. To accomplish that a fixed number of cards are assigned to the production line that accompany parts through the system. Similar to CONWIP the material flow in kanban is also regulated with cards. However, in kanban, cards are allocated to each station rather than the entire line. Cards are attached to parts and removed when they move from one station to another. Therefore, the number of cards at a station determines the maximum WIP level for that station.

Recently, there has also been some research on hybrid control policies that combine the strengths of the aforementioned pure control policies. Hodgson and Wang [31] simulate a multistage production system and conclude that the best result is obtained when using push policy at upstream stages and pull strategy at other stages. Altug [1] analyzes the performance of both the pure and mixed operating disciplines for four different production lines and finds out the best performing policies for each case.

1.2 Trends in Technology

In the last decade, technological advances particularly in the area of computer technologies have had profound effects on manufacturing. In this period the performance-to-price ratio of computers has roughly doubled every 18 months [84] and continues to do so. The introduction of low-cost computers and embedded controllers have led to new manufacturing concepts and methods such as computer integrated manufacturing (CIM) [52,78] and flexible manufacturing [76]. As a result of the proliferation of embedded controllers, a new generation of computer-controlled machine tools [79] have emerged which are capable of optimizing and controlling their own operation. These intelligent tools allow manufacturers to produce more complex and higher quality parts in shorter cycle times. The next logical step in manufacturing automation is to integrate the individual smart tools, that is, the islands of automation, to obtain large-scale factory automation [55]. In that case, the automation will not

be limited to individual machines and processes but will be expanded to encompass the entire manufacturing operation.

System integration [62] deals with the coordination of distributed intelligent agents to obtain higher efficiency and flexibility. In this concept, various intelligent devices in the manufacturing environment such as computers, machine tools, controllers and data collection devices are connected to each other to form a distributed network.

While the breakthroughs in computer technology are the main driving force behind the intelligent manufacturing systems, it is the recent advances in the communication technology [57] that made distributed systems possible. As the manufacturing system components became more intelligent the need for information exchange arose [32]. The information exchange needed to be computerized so as not to create bottlenecks in the system. The communication of intelligent devices in a manufacturing system is accomplished over a local area network (LAN) [50]. Essentially, all the devices in the factory that communicate over LAN form a manufacturing computer network [81].

1.3 A New Paradigm: Distributed Control

The technology has changed the way manufacturing is done. Today, the manufacturing plant is not a facility that houses a collection of manually operated machinery but a network of smart tools from a multitude of vendors with data processing capabilities. These self-aware tools and machines are also capable of exchanging information. They can inform others in the network about their status or receive commands and data from

other nodes. In this sense, each workstation can be regarded as an autonomous agent [47] in a distributed manufacturing environment, thus making the manufacturing system a multi-agent [24] system. This opens up new possibilities for manufacturing operations control.

1.3.1 Manufacturing Environment: A Complex System

A complex system is defined as a collection of autonomous components (agents) whose interactions determine the state of the overall system. It is generally believed that complex systems are beyond direct control [80]. A complex system operates through the continuous interaction and cooperation of its components, which may have independent and possibly conflicting interests. The chaotic behavior of such a system resulting from the numerous interactions between its components makes global optimization and control very difficult [29]. As stated by Ferber [24], "Any modification to the initial conditions, however infinitesimal, any introduction of random variables, however restricted, has effects which are amplified by the interactions between the agents and it is not possible to know in advance the precise state of the agents after a certain time has elapsed. Phenomena which are known as *butterfly effects* (a butterfly flapping its wings in Asia may produce a series of amplifications which lead to a tornado in North America) interfere with any precise forecasting and cause anomalies in results." In a complex system, analyzing and interpreting data in real time and taking proper actions becomes a difficult task because of the enormous amount of data available and also

because the cause-effect relationships in the system are not easy to determine.

Perhaps the most important consequence of a collection of cooperating agents is the so-called "emergent system behavior". Emergent behavior refers to those system properties and system dynamics that arise from the numerous modes of interactions among relatively simple components [4,5,39]. The example of an ant colony is often cited as an example [13,21]. Although each ant has limited data processing capabilities, the ant colony displays very complex behavior. The colony builds nests, collects and stores food, defends the nest, relocates in case of a flood, propagates, etc. It is well accepted that such complex behavior cannot be attributed to any given component of the system, but rather emerges from the interactions of the components.

Emergent behavior gives rise to another important aspect of studies into intelligent systems. As complexity is not assigned to a given component, the analytical methods, which seek an understanding of the system by studying its components, often fall short. Moreover, it becomes difficult to obtain predictive computational theories of emergent behavior, at least with the current set of conceptual and mathematical tools at our disposal. Studies of emergent behavior often lead to explanatory theories rather than predictive ones. Moreover, the studies are often descriptive with somewhat qualitative aspects, rather than the computationally mature formulations, such as queuing models. Nonetheless, it is possible to apply the cycles of the scientific method, that is, observation, hypothesizing,

experimentation, revision, to studies of emergent behavior. As mentioned, one would expect explanatory descriptive theories (e.g. as in the theory of evolution) rather than a more computationally precise theory.

As the systems we deal with get more complex, unpredictable and dynamic, the need for alternative control methods becomes more obvious. Distributed control appears to be a viable alternative to global (centralized) control. Unlike centralized control that requires the modeling of the entire system, distributed control models the behavior of individual units (agents) with well-formed, relatively simple rules. This provides efficient and flexible interactions between the components of the system. Therefore, it is believed that the distributed approach has lower overhead and is better suited for complex systems. One conclusion is that decentralization is a highly appropriate form of organization for complex distributed computer systems [27].

1.3.2 Distributed Manufacturing Operations Control

Traditional control approaches do not sufficiently address the important problems of the real-world manufacturing environment such as uncertainty and complexity [70]. Not only do they lack the ability to operate in dynamic environments, but also are not equipped to utilize the extensive amount of information present in today's manufacturing environment.

The concepts of CIM such as autonomous agents and cooperation of agents can be extended to develop a new generation of production control policies based on the

interaction of agents and local decision making. The intelligent components comprising the manufacturing system, by processing the information available in the system, can collectively determine the production pattern of the entire plant. Each unit in the system can determine its own production schedule. For instance, a machine tool may decide not to process a part even if it is not blocked or broken down. It does this to reduce the build-up in its inventory or perhaps to reduce congestion in one of its downstream stations. By creating a distributed network of autonomous devices capable of decision making, the burden of managing the entire manufacturing operation will shift from managers and engineers to the intelligent components. This would be the next step of factory automation.

Numerous examples are given in Section 1.3.4 that cover a wide range of applications of multi-agent systems. All of these applications are based on the idea of distributed intelligence and control. Then, it is fair to ask questions such as "What makes distributed systems so popular and preferred over centralized systems?," or "Why is there a trend towards multi-agent representation of systems and problems?," or "Why is there such an insistence on a local point of view?" These are legitimate questions that need to be addressed.

An important reason is the complexity of the problems faced. As the problems get more complex, it becomes more difficult to analyze them as a whole and obtain predictive solutions. In most cases, closed form solutions do not even exist. In others, the centralized approach becomes too computationally expensive and thus infeasible. We can take chess

as an example. Theoretically, it is possible to determine the winning strategy considering all possible moves, but in reality this proves impossible due to the enormous size of the state space (approximately 10^{120}). A method based on local approaches however might yield a good enough strategy that works rather quickly. Each piece can be modeled as an agent whose behavior is based on a heuristic.

Moreover, the complex behaviors of many systems result from the interactions of their individual components, which cannot be modeled. The complex behavior the system exhibits does not necessarily mean that its little pieces are governed by sophisticated mechanisms. Each little piece of the whole might behave following very simple, even myopic rules, that is, without having a global knowledge of the system to which it belongs. However, the interaction of these many pieces lead to an emergent complex behavior. In this case, using an approach based on a local point of view might be the only possible solution.

Another point that favors the distributed approach is the distributed nature of most problems. For example, the elements of a manufacturing plant (workstations, transporters, etc.) are physically and logically distributed. Moreover, these elements are already autonomous, most already have built-in data processing capability and are ready to interconnect.

Most importantly, the technologies needed for distributed control such as efficient and powerful embedded controllers and communication networks and protocols are available at low cost today. In essence, the emergence of these new technologies is

the main driving force behind the multitude of applications seen today.

Finally, the empirical evidence suggests that the distributed approach works. There are many real-life examples of distributed systems which perform successfully even though there is no analytical proof that they will.

Advantages of distributed manufacturing networks include the following:

- scalability. The same control structure can be used for production lines with different number of stations and different parameters.
- modularity. Workstations can be added or removed from the manufacturing environment without any disturbance or modification to the system or MOC.
- flexibility. If process parameters change, the system can adapt itself, there is no need for re-design.
- robustness. Malfunctions can be detected by the components of the system, malfunctioning unit can simply be taken out of the loop without interrupting the production.
- simplicity of control logic. Operations of agents are based on simple rules, much simpler than global control.

1.3.3 Technological Aspects

All the technological tools are available today for the development of a distributed manufacturing system capable of planning and controlling its own production:

- embedded controllers

- local area networks (LANs)
- artificial intelligence
- object-oriented modeling

Embedded controllers are without a doubt the most important element of manufacturing control and automation. They are what makes machine tools autonomously intelligent. Most microprocessors manufactured today end up as embedded controllers rather than being used in general purpose computers. This fact is a good indication of the extent of their impact in manufacturing and in our lives. The influence of embedded controllers in our lives goes beyond their use in manufacturing; they are a part of the appliances we use every day such as cameras, copiers, anti-lock brakes, and so on. The evidence that embedded controllers will continue to be a significant player in engineering is visible from the trend in embedded controller sales and use.

A local area network (LAN) is a system of hardware and software that allows the devices connected to the network to communicate over distances of several feet to several miles. Even though computer networks were initially developed for the communication of computers in the scientific community and later for the telecommunication industry, lately they have found use in manufacturing. Initially, the lack of a unified communication standard constituted a major obstacle to the widespread use of computer networks. However, the establishment of the IEEE-802 standard [48] prompted the vendors of automated equipment to design devices that comply with the standards. A General Motors

task force established in 1980 designed a communication protocol specifically for factory automation called manufacturing automation protocol (MAP) which is based on IEEE 802.4 Token Bus standard. As a result, manufacturing equipment by different vendors can now easily be interfaced simplifying the design of manufacturing networks and increasing their effectiveness.

In recent years, the controller area network (CAN) [10,11], a communication protocol developed by Bosch GmbH has become very popular in industrial automation. Even though it was initially developed for the automotive industry, it is used in various applications due to its robustness, reliability and high speed that makes it suitable for real-time applications.

Learning is an essential feature of adaptive systems. In recent years there has been an impressive body of work on artificial intelligence and its applications. Knowledge-based (expert) systems [6] and neural networks [35,77] are two primary branches of artificial intelligence.

Object-oriented modeling is a necessary tool when designing and analyzing systems consisting of many independent entities which interact with each other. It is based on identifying the independent objects in a system, defining rules and methods that govern their operation and providing a reliable structure for their communication with other objects in the system. Therefore the object-oriented approach is best suitable for modeling distributed systems [53]. It is the most suitable software technology for the development of simulation environments for the agent-based systems.

1.3.4 Related Work

A flurry of scientific activities in the field of multi-agent systems are evident from the international conferences such as the DAI (Distributed Artificial Intelligence) Workshop in the United States, MAAMAW (Modeling Autonomous Agents in Multi-Agent Worlds) in Europe and MACC (Multi-Agent and Concurrent Computing) in Japan. Recently, these three conferences have come together to create an international conference, ICMAS (International Conference on Multi-Agent Systems) which was held in San Francisco in 1995, Kyoto in 1996 and Paris in 1998.

Research in the area of complexity has been led by a think tank known as Santa Fe Institute in California, which is mainly focused on creating a common theoretical framework of complexity to understand the spontaneous, self-organizing dynamics of the world. The Santa Fe Institute brings together renowned researchers from various fields such as physics [16], economics [3], computer science [2,33], and study of evolution [40,56].

The idea of distributed systems first appeared in the area of information processing [14,26]. It has also found extensive use in general problem solving. In the broadest sense, problem solving involves computational (software) agents that have no physical structure. As stated by Ferber [24], distributed problem solving can be grouped into three different categories: distributed solving of problems, solving distributed problems and distributed techniques for problem solving.

Distributed solving of problems requires the cooperation of a number of "specialists" from different areas of expertise to solve a problem. In this case, the expertise is distributed among the agents, none of which possesses all the skills required to complete the job. The examples in literature include medical diagnosis [43], the design of an industrial product [7,37], fault finding in nets [38], the recognition of shapes [18], the understanding of natural language [59], or the control and monitoring system for a telecommunications network [83].

Solving distributed problems refers to applications that involve the analysis and control of physically distributed systems where it might be difficult to obtain a centralized overall view. In this case, it is the problem that is distributed, the agents can have similar skills. An example is the control of a communications or an energy network. The network itself is a distributed system. Control can be accomplished by decentralizing the monitoring tasks within the nodes of the network. Another example is the DVMT (Distributed Vehicle Monitoring Test) developed at Massachusetts University [44] that contributed a great deal to the field of Distributed Artificial Intelligence (DAI). In the DVMT project, a large array of sensors transmit traffic data to processing agents, which based on these data, try to identify and follow the vehicles, and obtain a picture of the current road traffic situation.

The work by Huberman and Clearwater [36] demonstrates a solution to thermal resource distribution in a building using a market-based system. In this system, computational agents

representing individual temperature controllers bid to buy or sell cool or warm air. This computerized auction is moderated by a central computer auctioneer.

Fisher et al. introduce the MARS system [25] which models cooperative scheduling within a society of shipping companies as a multi-agent system. In this model, the trucks of several geographically distributed shipping companies constitute the agents of the system. They are responsible for making local decisions about transportation. In MARS, the solution of the global scheduling problem emerges from local decision making and problem solving strategies. MARS is based on an extension of contract net protocol [30,68,69] developed by Smith. Contract net is a general negotiation protocol for communication and control among cooperative agents, with one agent announcing the availability of tasks and awarding them to other bidding agents.

Multi-agent approach can also be applied to solve problems where, unlike in previous cases, neither the problem domain nor the expertise is distributed. This application of multi-agent approach is classified as distributed technique for problem solving. Allocating resources to tasks, or stacking cubes are two of the examples. In these problems, resources and tasks, or cubes can be considered as agents which operate according to certain constraints. The works by Ghedira [28], and Liu and Sycara [46] demonstrate the success of multi-agent approach to solve constraint-based problems.

In addition to problem solving, multi-agent approach also shows great promise in the area of robotics. A team of robots can be used to perform tasks such as exploring a planet,

monitoring buildings, doing repairs, which might be impossible or potentially dangerous for humans.

Research in this area originates from space exploration efforts of NASA. When NASA decided to explore the planet Mars, it quickly became evident that the initial phase of exploration would be carried out with the help of robots. It became also quite clear that the robots would have to be autonomous since radio transmission from Earth to Mars takes tens of minutes, too long for remote control. The idea of using a large number of inexpensive robots with limited capability for this purpose is first suggested by Brooks [9]. This of course requires the coordination of robots, a problem for which the multi-agent approach is particularly suitable. Deneubourg et al. [19] and Steels [72] made significant contributions in this area by demonstrating that it is possible to perform the necessary tasks by using reactive [20] robots that are programmed to behave in a way similar to ants. Another example to the use of multi-agent systems in exploration is the work done by Maio and Rizzi [49] who propose an unsupervised exploration algorithm in which several autonomous agents cooperate to acquire knowledge of the environment.

Coordination of moving vehicles is also a popular research topic. Specifically, the formation maintenance and control of mobile robots has drawn recent attention. Wang [82] develops a strategy where individual robots try to maintain their positions relative to a leader or another robot. Chen and Luh [12] demonstrate formation generation by distributed control. In their work, large groups of robots are shown to cooperatively

move in various geometric formations. Balch and Arkin [8] take this one step further by integrating obstacle avoidance and other navigational behaviors with formation behavior of robots. A variation to this problem is agent tracking which involves observing and predicting an agent's behavior and movement. Agent tracking has extensive applications in military (air-combat) [74] and air-traffic control [87].

In the area of manufacturing there has been a fair amount of work done on distributed systems from the perspective of system integration. An example of that is the CCE-CNMA project [22], which deals with the integration of high-technology applications and specifically the development of open standards for communication between factory automation applications. McGehee et al. [51] demonstrate a large-scale application of CIM which includes the integration of the processing equipment with all the supporting systems for product and process specification, product planning and scheduling, and material handling and tracking in semi-conductor wafer fabrication. In the field of industrial process control, the Flavors Paint Shop system [54] used in the painting of lorries presents an example for an industrial application of a multi-agent system. This application involves the optimized use of a limited number of paint stations. The centralized approach employs a classic program, which is very expensive and hard to maintain. It also requires very detailed planning with a small margin of error. In the multi-agent approach, each painting station is represented as an agent that operates based on simple rules. In spite of its simplicity, the distributed approach results in a drastic

reduction of paint-change operations which saves more than a million dollars a year.

In addition, agent-based models have been proposed for distributed scheduling of tasks where the coordination of the material flow occurs with the cooperation of agents. In these models, agents represent physical entities such as parts and resources (workstations). Sikora and Shaw [66] introduce an agent-based framework for manufacturing systems and present a real-world application. Lin and Solberg [45] propose a scheduling procedure in a distributed control environment based on the negotiations between parts and machines. Other work done in the area of distributed scheduling includes Sadeh [60], Shaw [63], and Sycara [73].

1.4 Motivation

Generally speaking, current industrial engineering practices continue to rely heavily on traditional methods for production control. These methods may be characterized by two common points: a unicentric view of the physical plant, and the treatment of the system components as passive elements, that is, uninvolved in the decision making process. It is actually the latter that gives rise to the traditional worldview. Both, the familiar policy-based (e.g., push, kanban, conwip) and solution-based (e.g., mathematical programming) techniques share this worldview. Quite naturally, it is the technological advances affecting the latter issue that encourages the re-visitation of the current worldview of manufacturing control techniques, and gives rise to this study.

In general terms, economic factors favor a set of distributed and networked controllers over centralized data processing. The human brain, for example is clearly built as a network of smaller elements (neurons), which individually have very little capability. Similarly, elements of manufacturing systems are becoming "smarter," that is to say, include more and more embedded control. For example, almost all machine tools manufactured today contain data processing capabilities at a level of at least a standard PC. Moreover, most machine tools are now connected to some type of communications network. Embedded control and network connectivity allow elements of manufacturing systems to make local decisions based on dynamic local data, store the data (keep its history), and communicate the data with other elements. In such an environment, the control of a manufacturing system should no longer be limited to the generation of a master schedule, sent to the elements for execution. Rather than a top-down control, it seems reasonable to allow each element is to make local decisions and "deduce" its own role in a given system task.

There is a need and opportunity in industrial engineering to investigate and develop approaches to MOC, driven by these emerging technologies. In a larger sense, the networked/embedded data processing paradigm may affect many other systems in which industrial engineers are interested, such as resource planning, transportation, facilities planning, and quality control. We limit the scope of this study to MOC for brevity.

1.5 Approach

This study is a scientific experiment to investigate the feasibility of one possible alternative to manufacturing operations control driven by the technological changes. It should be noted at the outset that this study makes no claim to develop the terminal solution to manufacturing operations control, or even if the proposed approach will ever become industrially feasible. Rather, it acknowledges the ever-changing nature of engineering practices, which, at least in many industrial engineering systems, often evolve within industry, and which subsequently are studied by academia. In this respect, the current study is arguably one of the very first academic efforts to rise the prospect of technology-induced alternatives to the current manufacturing control paradigms.

We use the scientific method and start with a hypothesis:

> *It is feasible to develop cost-effective engineering solutions to manufacturing operations control based on a networked set of manufacturing elements with embedded data processing capabilities.*

We propose to use low-cost embedded controllers and a very simple networking protocol. We hypothesize that such a system can achieve the efficiency and effectiveness of commonly used contemporary manufacturing operations control techniques. The use of low-cost embedded controllers implies that the solution techniques we adopt will require little data processing

capabilities (in the order of 10 to 100 MIPS, or roughly, at a cost of $10 to $100). This being an engineering study, we will attempt to demonstrate the validity of our hypothesis by actually developing an engineering implementation. We recognize that the feasibility of the approach hinges on, in no little part, the choice of technologies and protocols.

We take a minimalist approach. That is, since our objective is to investigate the feasibility of distributed manufacturing operations control, we would like to demonstrate such feasibility for the more simple cases, focusing on the fundamental ideas rather than on the details. We take a production line arrangement and compare the performance to the more commonly used techniques such as push and pull (kanban).

Our approach involves two phases. First we study several candidate architectures. We make use of analytical as well as empirical tools to identify the more promising architectures. The second phase involves an actual engineering implementation.

1.6 Organization

Chapter 2 contains a brief discussion of the technologies necessary for DMOC, namely, embedded data processing and network communication technologies.

A framework for an adaptive, distributed manufacturing system with autonomously intelligent stations is presented in Chapter 3. The key elements of such a system are identified, the manufacturing setup used in the simulations is described and the control strategies and learning mechanisms employed are explained.

Chapter 4 deals with the simulation of the distributed manufacturing system. First, five different simulation scenarios used in the simulations are introduced. Then, the simulation results of static and adaptive control policies are presented and compared. Chapter 4 also includes a discussion on adaptive behavior in general. The learning pattern of the system is examined under various conditions and the key elements of learning are identified and analyzed.

Chapter 5 focuses on the next phase, namely the physical implementation of the manufacturing system modeled. A serial production line consisting of four networked embedded controller units is designed and physically implemented. Each controller unit in the emulated production line represents an autonomously intelligent workstation. The key design issues are discussed and implementation problems resulting from hardware and software limitations are investigated. Synchronization of autonomous stations, construction of a suitable message structure, determination of a reliable communication speed and message frequency, error handling, memory requirements, and system reliability are some of the issues addressed.

In Chapter 6, an overall evaluation of distributed manufacturing operations control (DMOC) is presented and the future of DMOC in industrial engineering is discussed.

CHAPTER 2
ELEMENTS OF DISTRIBUTED CONTROL

In layman's terms, distributed control refers to a system with several networked processors. Each node in the network has the capability to collect local information, store it and process it. In other words, the burden of data processing is distributed over an entire network rather than being assigned to a single computer. Decentralized data processing is typically implemented by some type of a microprocessor at each node. Data collection and storage implies that the nodes are equipped with memory. In addition, input/output (I/O) devices are necessary for the node to communicate with the external world and other nodes in the network. The processor, memory and I/O devices need to be arranged into a compact package small enough to fit into the device to be controlled.

Another requirement of distributed control is the capability of the distributed components to communicate with each other. Thus, the components of the system form a communication network.

2.1 Embedded Data Processing

Owing to the advances in microprocessor/microcontroller technology in the last two decades, today a one-chip solution

exists. As a result of the continuous process of miniaturization, all of the components necessary are built right onto one chip. A microcontroller is a highly integrated chip, which includes, on one chip, all or most of the parts needed for a controller. These include the following:

CPU (central processing unit)

RAM (random access memory)

EPROM/PROM/ROM (erasable programmable read only memory)

I/O (input/output - serial and parallel)

analog-to-digital converters

timers

interrupt controller

An embedded controller is, as its name suggests, basically a microcontroller embedded in the device that it controls. Unlike personal computers (PC) that are general-purpose data processing devices, embedded controllers are designed specifically for the control application for which they are intended. As opposed to a PC that can run a wide variety of application programs (word processor, spreadsheet, graphics, compiler, and so on), an embedded controller runs a single program developed to control a specific device.

Most often embedded controllers must meet special requirements such as cost-effectiveness, low power, small footprint. This is mainly why the processing power of an embedded controller is usually less than that of a PC. Cost is also kept low by including features only specific to the task. In a sense,

embedded controllers are specialized low-cost bare-bone computers. In contrast to PCs, embedded controllers do not use standard input/output devices such as a keyboard, a mouse and a monitor. They mainly react to external signals they receive from sensors and possibly other smart devices. The input to a controller, therefore, is generally an electrical signal (analog or digital) which likely triggers an event (interrupt) and causes the controller to activate a specific program routine called an interrupt service routine (ISR). The controller responds to the input by sending a control signal to the device that it controls.

Due to the fact that they are not intended to be used as general-purpose computers, and also because of their limitations in processing power and memory resulting from low-cost requirement, embedded controllers rarely employ a standard operating system. This has very important implications when it comes to programming them [42]. No operating system means that the programmer will have to worry about things that normally would not be dealt with when working with a PC. PC programmers are used to having an operating system handle such mundane things as I/O, memory management, time management, error processing, inter-device communications, and so forth [61]. In a development environment where there is no operating system, even storing data might become a challenging task. While opening a text (ASCII) file to write data is no problem in, say, Unix, doing the same in a microcontroller is not trivial. Even a task as simple as reading data from the keyboard might require additional effort

for a programmer programming a microcontroller while all compilers developed for PCs have standard input/output routines a programmer can use without effort.

In embedded applications, memory limitations often force programmers to program in assembly language [85]. Assembly is a low-level programming language that generates much tighter code than high-level languages such as Pascal and C [64]. However in general, coding in assembly requires better programming skills. Moreover, debugging is significantly more difficult and at times can become overwhelming. For that reason assembly programs need to be compact. A feature common to most embedded applications therefore is that they have little overhead.

The use of embedded controllers is not limited to just manufacturing. Today, embedded control is such an integral part of our lives that embedded control products can be found in all market segments: consumer electronics, PC peripherals, telecommunications (including fast-growing personal telecommunication products), automotive and industrial. Specifically, the automotive market is the major driving force in the embedded controller market, especially at its high end. Several microcontroller families were developed specifically for automotive applications and were subsequently modified to serve other embedded applications. The steady increase in the use of microcontrollers as embedded controllers in automobiles is evident in Table 2-1.

Table 2-1. Average Semiconductor Content per Passenger Automobile (in dollars).

'91	'92	'93	'94	'95	'96	'97	'98	'99
634	712	905	1,068	1,237	1,339	1,410	1,574	1,852

Source: ICE

Table 2-2. World Wide Microcontroller Shipments (in millions of dollars).

	'91	'92	'93	'94	'95	'96	'97	'98	'99
4-bit	1,597	1,596	1,698	1,761	1,826	1,849	1,881	1,856	1,816
8-bit	2,615	2,862	3,703	4,689	5,634	6,553	7,529	8,423	9,219
16-bit	303	340	484	810	1,170	1,628	2,191	2,969	3,678

Source: WSTS & ICE

Table 2-3. World Wide Microcontroller Shipments (in millions).

	'91	'92	'93	'94	'95	'96	'97	'98	'99
4-bit	906	979	1,036	1,063	1,110	1,100	1,096	1,064	1,025
8-bit	753	843	1,073	1,449	1,803	2,123	2,374	2,556	2,681
16-bit	38	45	59	106	157	227	313	419	501

Source: WSTS & ICE

However, automotive industry is not the only market that is growing. Consumer electronics is also a booming business. It has been reported that by the year 2000 there will be 240 microcontrollers in the average North American's home. Table 2-2 and Table 2-3, which clearly show the growing microcontroller sales in this decade, might give an idea about the size of the overall embedded controller market today.

2.2 Network Communications

A very broad definition of a communication network is a system of interconnected devices capable of exchanging information. In the context of this study however, a more restricted interpretation is used. The components of the network should be intelligent (computerized) devices that can be programmed to engage in communication with any other device in the network in an autonomous fashion (i.e., without the assistance of humans).

In recent years, there has been much interest in interconnecting computers in a network [57]. One reason is that a network of computers presents a cost-effective alternative to a more powerful central computer. Not only does distributed processing offer more flexibility to individual users, but also compared to centralized processing, it proves to be more reliable and resistant to system failures. As a result, computer networks have found extensive use in office and factory automation. Local

area networks (LAN) that interconnect computerized elements locally, that is, within a few miles, are commonplace today.

Traditionally, networks have been used to increase the processing power of computers (i.e., multiprocessing), electronic mail, sharing expensive resources (e.g., laser printers, expensive programs), file transfer, and so on. However, these applications in general support an office or laboratory environment. From a manufacturing point of view, there has been a growing interest in applying networks in a manufacturing environment recently. This interest has been fueled by the realization that automation is a key factor to improving productivity and that network communications is essential to industrial automation.

Local area networks are the most commonly used networks in manufacturing automation. Typical characteristics of a LAN are the high data transfer rate, reliable transmission with low error rates, and support for full connectivity. As shown in Figure 2-1, the most common topologies for LAN are bus, ring and star.

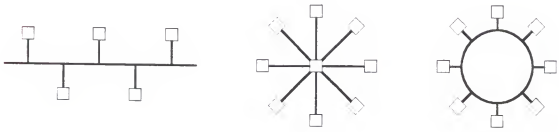


Figure 2-1. Most common LAN topologies.

For the communication to take place between the components of a network, it is not enough to simply physically connect the devices to the network. To ensure and coordinate the information flow, certain rules must be established. For instance, the physical medium (e.g. coaxial cable, twisted pair, fiber optic, etc.), through which the data transfer occurs, must be defined. The access to the communication medium (medium access control) must be regulated. An error detection and correction mechanism must be implemented. Questions such as "How do nodes exchange messages?," or "What is the format of messages?," or "How does a node get access to a shared communication medium?," or "How do nodes deal with transmission errors?" need to be answered to ensure efficient and reliable transfer of information. Obviously, there are many possible answers to the questions above. A set of answers and solutions, when grouped together, form a protocol. A communication protocol is basically a set of rules and methods that provides all the design details and dictates how communication occurs in a network. Ethernet (CSMA/CD), token passing bus, token passing ring are some examples to existing communication protocols.

The development of standards is essential to the acceptance and spread of new technologies. With the increasing use and popularity of communication networks, it has become necessary to interface networks that operate based on different protocols. To promote compatibility of network designs and protocols, the International Organization for Standardization (ISO) proposed an

International Organization for Standardization (ISO) proposed an architecture model called the Open Systems Interconnection (OSI) reference model [17]. As depicted in Figure 2-2, OSI is a layered architecture, consisting of seven layers. The idea behind the layered architecture is to divide the system into independent segments (layers), whose internal workings are not known to other

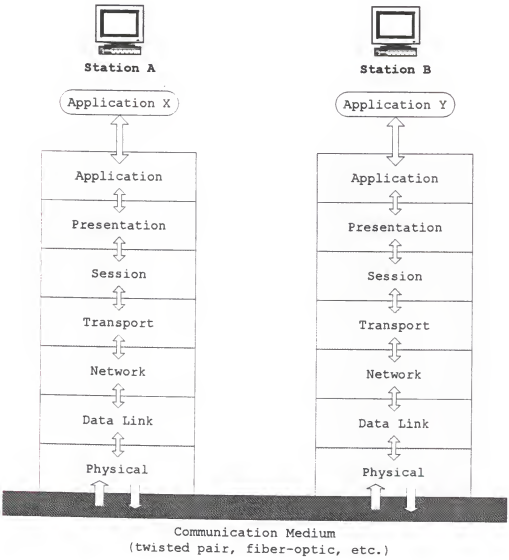


Figure 2-2. Message transfer between two nodes in the network in OSI model.

segments. This concept is also adopted by object-oriented programming. Each layer represents certain tasks to be performed for successful transmission and reception of messages. These tasks (services) include encryption of messages, partitioning of messages into smaller packets, routing decisions, error detection, medium access control, and actual transmission of messages via electrical signals. Each layer adds to the services provided by the lower layers so that the highest layer (application layer in Figure 2-2) is provided a full set of services to manage communications and run distributed applications.

CHAPTER 3
MODELING OF A DISTRIBUTED MANUFACTURING SYSTEM

3.1 Overview

As stated before, the objective of this study is to demonstrate that it is technologically possible and economically feasible to apply the principles of distributed control to manufacturing operations.

The driving force behind this study is twofold. The traditional formal methods of industrial engineering are becoming less sufficient in addressing the needs of today's highly complex manufacturing environments. Furthermore, due to technological innovations in recent past the manufacturing tools and components have become smarter (self-aware) and more autonomous.

It is our belief that with the recent advances, the technology has reached a point that it can support the concept of distributed manufacturing control. As discussed in Chapters 1 and 2, examples of distributed control can already be seen in many areas of engineering from automotive and textile industries to large optical telescopes. Our goal is to show that it has potential in the area of manufacturing operations control and thus deserves some attention.

We take a minimalist approach to focus our attention on the fundamental ideas; we model and implement a simple serial production line with limited communication between stations. The production line consists of intelligent, autonomous stations where the control is achieved by the cooperation and communication of those individual stations that are capable of adapting and responding to the changing conditions of the environment.

The amount of communication between stations is limited to reduce the complexity of the model. A station can send messages only to its upstream and to its downstream neighbor. In addition, only a few message types are allowed. For the same reason a simple set of rules are selected that characterize the behavior of stations. Also, a rather modest learning mechanism is implemented to achieve adaptive behavior.

In the next section, a description of the manufacturing environment is presented which includes the physical arrangement of the stations, the mode of operation, the definition of the overall and local goals, as well as the defining features of an adaptive distributed manufacturing system.

A detailed explanation of the heuristics responsible for the intelligent behavior of the manufacturing system is given in Section 3.3.

3.2 Manufacturing Environment

3.2.1 Setup

As stated in the previous section, a production line of serially connected stations is selected as the basis for the distributed manufacturing system to be modeled and implemented. As Figure 3-1 shows, stations are separated with variable size buffers. Each station controls the buffer between itself and its upstream neighbor. It is assumed that an infinite supply of parts is present that can continuously feed the line with parts. The parts enter the production line at one end, are processed sequentially by the stations and exit from the other end as finished products. Only one type of product is manufactured and the order of operations is fixed. Each station is capable of performing one specific operation on the part that the upstream station places in its buffer. Stations use the first-come-first-serve dispatching rule when selecting the part to process. Conceptually, this is a simple production line that offers no flexibility in terms of scheduling and routing.

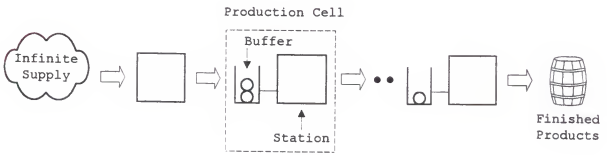


Figure 3-1. Serial Production Line.

3.2.2 Production Goal and Mode of Operation

The main objective of the manufacturing system modeled is to maximize its profit which is defined in terms of a reward for finished products and an inventory cost due to the accumulated inventory in the station buffers.

The system is rewarded for every finished product. On the other hand, there is a cost associated with every part that sits in a station buffer waiting to be processed. The goal is to ship out as many products as possible with the least amount of work-in-process (WIP) inventory.

In a production line, the difference in processes performed by stations, the fluctuation of service times, or unexpected station break-downs disturb the continuous flow of materials through the system causing "hot spots." Parts accumulate in the buffers of stations that become the "bottlenecks" of the system. The resulting high inventory levels drive the cost up making the whole system less profitable. A control strategy is necessary that will temporarily shut down or slow down the upstream stations to dissipate the accumulated inventory.

If the inventory cost is very high, it will become the dominant factor and each station should slow down the production at the upstream station to keep its inventory at minimum. The savings from the reduced inventory cost will offset the loss of reward due to the low production output. On the other hand, if the inventory cost is negligible, the production line should operate like a true push system with high levels of WIP. This

will result in the maximum production count and the reward will make up for the cost resulting from the excessive inventory build-up. Essentially, the system tries to find a balance between inventory cost and production reward. It constantly searches for the operating point that represents the highest profit possible.

In the production line model presented here, the control of material flow is distributed to individual nodes (station). Each station interacts with its upstream and downstream counterpart. A station is not allowed to unload a processed part unless it is granted permission to do so by the downstream station that controls the buffer. Hence, the actions of a particular node affect the status and the performance of other nodes as well. For instance, if a station (node #N) does not accept the processed part from its upstream neighbor, then the upstream station (node #N-1) becomes blocked. Similarly, the same action may later cause the downstream station (node #N+1) to starve.

In essence, our production line is a pull system with dynamic local control. It can also be described as a system with occasional delays to the flow of parts at bottleneck nodes.

3.2.3 Features of the Manufacturing System

The distributed manufacturing system to be modeled and to be implemented has the following properties:

Distributed control. The distributed nature of the manufacturing system results from the fact that each station in the production line is regarded as an intelligent autonomous agent that has decision making capabilities. As a whole, this is a goal-driven

multi-agent system where each station works as an autonomous entity trying to maximize its own profit. The decisions and actions of each station are geared towards obtaining their own best interest. The physical point of control is at station buffers. Each station controls its own buffer and can prevent its upstream counterpart to unload the processed parts into its buffer. If a particular station determines that the inventory level in its buffer is already high, it might decide not to allow any more parts into it until the inventory level drops, thereby reducing the inventory cost. Thus, the flow of parts through the line is strictly controlled by the selfish behavior of individual stations. Agents work only for themselves, not caring for the well-being of others in the system. This approach simplifies the control strategy implemented in each node because the amount of interaction and negotiation between stations is reduced. The modular nature allows a lot of flexibility; the production line can be expanded by simply inserting more nodes with minimal modification to the system. Coupled with the fact that learning is incorporated into it, this is a simple but effective control mechanism that provides adaptive, dynamic control in a distributed fashion.

In our model, agents use a rule-based approach in decision making. Decisions made by each agent (node) are based on the evaluation of the status of that particular node described by local parameters. In other words, each agent "looks around," determines its current state based on what it sees and takes action by selecting one of the rules in its rule base.

Rule-based decision making. A pre-determined rule set used at the nodes of the production line determines the material flow in the system. The rule-based approach provides a simple yet robust structure given the rule set does not contain overlapping and conflicting rules. The rules involve one or more local parameters such as the current status of the node (DOWN, WORKING, STARVING, etc.) and the buffer level and have the following format:

If (condition)

Then "action"

Here, the condition may be for instance

parameter1 == val1 AND parameter2 > val2

where parameter1 and parameter2 are system parameters and val1 and val2 are two specific values those parameters may have. The action is the answer, in the form of either permission or refusal, sent to the upstream station which wants to unload the part it just finished processing.

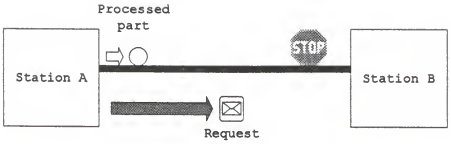
Obviously, the selection of local parameters and the criteria the nodes use to evaluate their performance, greatly affect how the individual nodes behave, and consequently, how the whole system performs. For that reason, two heuristics have been developed which are based on separate parameters. A detailed discussion of these heuristics are presented in the next section.

Inter-nodal communication. The operation and the behavior of the individual stations in our production line is determined by

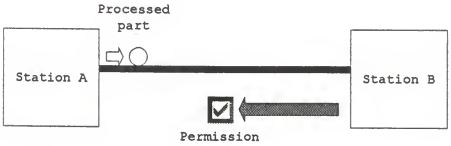
their interactions with their upstream and downstream stations. Even the relatively simple heuristics described in the next section require each node to communicate with other nodes in the system. Therefore, inter-nodal communication is an important aspect of our manufacturing system.

Figure 3-2 illustrates the part transfer protocol between stations. The communication is initiated when a station (A) is finished processing a part. After the processing is completed, that station (A) immediately sends a message to the downstream station (node B) requesting permission to place the finished part into the downstream station's (B) buffer. After evaluating the situation *B* decides whether it is in its best interest to accept the part or not. It either grants permission to *A* to unload the part or refuses the request. In the case of refusal, *A* becomes blocked and cannot process another part. *A* remains in contact with *B* until *B* grants permission and removes the part.

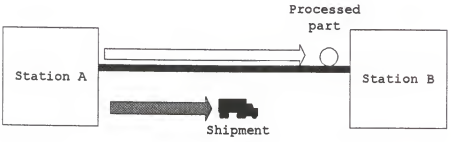
Learning. The rule-based decision mechanism that the proposed manufacturing system employs does not fully account for the disturbances the system may experience during the operation such as machine break-downs, fluctuations in service times, process and setup modifications, or changes in product orders. To make the system adapt to changing environmental conditions, a simple learning mechanism is added to the control structure. In accordance with the notion of distributed control, the learning is performed locally, that is, at individual nodes. At each node, a learning mechanism based on parameter calibration is incorporated into the decision process.



(a)



(b)



(c)

Figure 3-2. Illustration of part transfer protocol.
a) Station A requests permission to transfer.
b) Station B grants permission.
c) Station A unloads the part.

In specific intervals during the manufacturing process, the agents evaluate their performance in terms of profit and attempt to establish a pattern by comparing their costs and incomes to those in previous intervals. Based on their observations they modify their rule-base by adjusting certain rule parameters. The modified rule set is used throughout the next interval. In the long run, the parameters are calibrated such that the behavior of a particular agent fits the production pattern of the other agents with which it has been interacting.

Changes in the manufacturing process such as the modification of process times in stations, addition or removal of stations from the production line, increase or decrease in buffer sizes, or changes in production and financial policies such as higher emphasis on timely delivery, lower inventory costs, or higher production rewards can be accommodated by letting the system run and adapt to the new conditions.

3.3 Adaptive Control Policies

Two adaptive control strategies are presented in this section. These control strategies consist of two closely coupled components; a rule-based control heuristic that determines the dynamic behavior of an agent, and a learning mechanism that refines the rule-base modifying the behavior of the agent so that it can react to changing conditions of the environment.

The primary goal when developing the rule-based control heuristics is that they be described by simple rules. Each heuristic includes a control parameter, which constitutes the

link between the rule-based control heuristic and the learning mechanism. The agent evaluates its performance in every interval and modifies the value of the control parameter accordingly. Since the control parameter is included in one or more rules, this changes the rule set and consequently the behavior of the agent. Thus, the adaptive nature of the control strategies described here comes from the periodic manipulation of the control parameter.

The control strategies discussed below employ very similar rule sets. The main difference between them is the control parameter used and the way it is manipulated by the learning process. As explained in Section 3.2.2, the goal of each agent, where the control strategy is implemented, is to maximize its profit defined as,

$$\text{Profit} = \text{Production\_Reward} - \text{Inventory\_Cost} \qquad (3\text{-}1)$$

The distributed control is accomplished by each agent controlling the input to its buffer. Dictated by the rules in its rule set a station may or may not authorize the upstream station to place a new part into its buffer. Hence, the material flow in the entire system is collectively determined by the actions of the individual stations.

3.3.1 Control Policy Buffer_Size

In this control strategy, the *buffer_size*, that is, the number of parts a buffer can hold, is the control parameter. Each agent (station) is capable of varying the size of its buffer. Depending on how well it did in the previous interval, a

station may increase or decrease the size of its buffer, thereby controlling the maximum inventory build-up.

The rules that describe the dynamic behavior of the agents in this strategy are explained below:

Rule 1. If a station is *blocked* because its downstream station does not grant permission to unload, then it does not authorize its upstream station to unload. The station remains *blocked*.

Rule 2. If a station is *down* and is being repaired, then it does not authorize its upstream station to unload. The station remains *down*.

Rule 3. If a station is *starving*, that is, functional but not processing any part at the moment because its buffer is empty, then it authorizes its upstream station to unload. If part is loaded to the buffer the station becomes *up and running*.

Rule 4. If a station is *up* and *running*, that is, currently processing a part, then it authorizes the upstream station to unload if its buffer is not full. The station remains *up and running*.

In simple words, if a station is not functional because it is blocked or down, it does not accept new materials into its buffer. However, if it is functional, either starving or busy, it may allow new materials into its buffer. Therefore, the current inventory level and the *buffer_size*, which is controlled by the station, becomes the determining factor for the influx of

materials into the buffer. A station will reduce its *buffer_size* if the excessive inventory levels cause it to operate inefficiently. On the other hand, if the reward obtained from processed parts is so high, that it can tolerate high inventory levels, it will increase its *buffer_size* allowing parts to accumulate in its buffer. By doing so, it will guarantee continued operation even if its upstream counterpart breaks down and cannot supply new material for a certain period of time.

The learning module uses a heuristic that looks at the variations in profit in recent history. If an increase in *buffer_size* results in an increase in profit in the last interval compared to the previous interval then the *buffer_size* will be increased further. If, however, it results in a decrease in profit, the *buffer_size* will be decreased. In rule format,

Rule 1. If at the beginning of the last interval *buffer_size* had been increased, and as a result the *profit* went up, then increase *buffer_size*.

Rule 2. If at the beginning of the last interval *buffer_size* had been reduced, and as a result the *profit* went up, then reduce *buffer_size*.

Rule 3. If at the beginning of the last interval *buffer_size* had been increased, and as a result the *profit* went down, then reduce *buffer_size*.

Rule 4. If at the beginning of the last interval *buffer_size* had been reduced, and as a result the *profit* went down,

then increase *buffer_size*.

The simulations show that the learning mechanism described above has some weaknesses. In certain situations it cannot adequately determine the state of the system and therefore fails to change the *buffer_size* in the proper direction. This generally occurs when the upstream or downstream stations have substantial impact on the operation of a station rather than the change in *buffer_size* of that station.

Let us consider the situation where the *buffer_size* of a particular station (station B) is currently low and consequently the station must work with a low level of inventory. Let us assume that before a new interval the *buffer_size* is increased and at the end of that interval it is found out that the profit went down. According to the learning mechanism described, Rule 3 will fire and reduce *buffer_size* for the next interval. Let us examine the effectiveness of this decision.

When the *buffer_size* of station B goes up it is expected to process more parts. If that happens, its overall profit will go up if the reward obtained from producing additional parts is higher than the extra inventory cost due to the increased buffer levels. On the other hand, the profit will go down if the excess reward cannot offset the excess inventory cost. The fact that the profit went down in the situation described above might suggest that this is just what has happened. However, the drop in profit might have also been caused by a lack of production rather than excessive inventory cost. In other words, the production count might not have gone up even though it is what

was intended when *buffer_size* was increased. This can happen due to two reasons; the production output of the upstream station (station A) might go down due to unusually high break-down periods or process changes (longer service times, etc.), thereby reducing the material flow into station B causing it to process less parts, or the downstream station (station C) might block station B an unusually long period of time due to changes in its own production policies or parameters. In the latter case, the response of the learning heuristic (firing of Rule 3) is justified. However, if the former is true, then reducing *buffer_size* is clearly not the answer and will have just the opposite affect; it will lower the production rate further reducing the profit even more.

As demonstrated above the current learning mechanism might not make the right decision if the drop in profit is caused by the behavior of the upstream station. It can be improved if Rule 3 is expanded such that,

Rule 3a. If at the beginning of the last interval *buffer_size* had been increased, and as a result the *profit* went down and more parts have been processed, then reduce b*uffer_size*.

Rule 3b. If at the beginning of the last interval *buffer_size* had been increased, and as a result the *profit* went down and less parts have been processed, then increase *buffer_size*.

Rule 3a above deals with the case where the reward obtained from the production of additional parts cannot offset the cost caused by the increased inventory.

Rule 3b accounts for the possibility that the production output might go down even with a higher *buffer_size* due to lack of input from the upstream station. In this case the *buffer_size* will be further increased even though the profit is down.

Another case where the current learning mechanism might be inadequate is if the profit goes down as a result of *buffer_size* being reduced. Reducing *buffer_size* is expected to lower the inventory cost, but what if just the opposite happens and the inventory cost goes up? In that situation the learning mechanism above responds by increasing the *buffer_size* (Rule 4) which will likely increase the inventory cost even more.

The increase in the inventory cost might have been caused by the actions of the upstream (station A) or downstream (station C) stations and have less to do with the operation of this station (station B). One explanation might be that the upstream station increased its own *buffer_size* in the previous interval and thus processed and released more parts into station B's buffer thereby saturating B's inventory and driving the inventory cost high. Increasing *buffer_size* as Rule 4 suggests would lead to an even higher inventory build-up and make the station even less profitable. Another possibility is the downstream station blocking station B considerably more than it did in the previous interval. This would limit the number of

parts processed by B and thus reduce the profit. In both cases, the appropriate response by the learning mechanism should be to decrease *buffer_size*. These cases will be accounted for if Rule 4 in the original rule base is expanded as follows:

Rule 4a. If at the beginning of the last interval *buffer_size* had been reduced, and as a result the *profit* went down and a higher average inventory level has been achieved, then reduce *buffer_size*.

Rule 4b. If at the beginning of the last interval *buffer_size* had been reduced, and as a result the *profit* went down and a lower average inventory level has been achieved, then increase *buffer_size*.

In the control mechanism described above, the capacity of the input buffer is the control parameter. In other words, it is the size of the buffer that controls the flow of parts and consequently the production rate. One drawback of this method is that it does not allow the precise control of production. Buffer capacity can be increased or decreased only by an integer amount; the minimum change to the control parameter is one. Unfortunately, even this minimal change might result in a significant change in the production behavior. Therefore, *buffer_size* turns out to be a rather crude control mechanism. The addition of a fine-tuning mechanism to *buffer_size* is proposed in Appendix B and then analyzed using a Markov model. The objective is to generate a continuous profit function that is defined not only at integer values of buffer size, but also

between the discrete points. For example, the resulting model should allow an effective buffer size of, say, 3.5. To achieve this, delays are introduced to the flow of parts at the output of stations as a secondary control mechanism. In short, the analysis in Appendix B focuses on the possibility that the modified *buffer_size* might generate higher profit at the intermediate buffer size values.

In the Markov model developed in Appendix B, the part transfer delay is represented by a release probability. Due to analytical difficulties in modeling production lines with multiple workstations, a modest two-station one-intermediate-buffer arrangement is selected. For the sake of simplicity, exponential distribution is assumed for production parameters which is consistent with similar studies [23,75]. The analysis reveals that the suggested modification to the control mechanism does not produce the desired effect; it is shown that for the two-station arrangement described, it is impossible to achieve a higher profit at intermediate values of buffer size than at the discrete values.

It should be emphasized, however, that the Markov model presented in Appendix B has several shortcomings necessitated to achieve an acceptable level of mathematical tractability. It is very limited, in that it models a production line consisting of only two stations and does not give any insight how the performance would change if the number of stations is increased. The interactions between individual stations might change the emergent behavior of the system drastically. Due to its limited

scope, the proposed model does not account for that. In addition, the assumption that the parameters are exponentially distributed further restricts its use. This model is obviously not applicable if exponential distribution for production parameters is not acceptable. Another point to be made is that transmission delays, which are at the core of the model, are represented by a release (holding) probability in the model. It is not certain however how a specific amount of time can be mapped to a probability.

Due to these shortcomings, the Markov model presented in Appendix B does not precisely represent the desired control mechanism and thus it is not very useful in a real-world setting.

3.3.2 Control Policy Delay_Time

This control strategy is based on the principle that a production line can operate as fast as its slowest component. The average processing speed of stations differ from each other because each has a different service time, break-down rate and repair time. The difference in processing times cause excessive inventory build-up in front of slow stations. The objective here is to synchronize the operation of stations by introducing delays to material transfer from one station to another. In the case depicted in Figure 3-3 a slow station follows a fast station with average processing times PS_1 and PS_2. If the slow station waits for ΔPS where,

$$\Delta PS = PS_2 - PS_1 \qquad (3\text{-}2)$$

before accepting a new part from the fast station, their operation will be somewhat synchronized and excessive inventory build-up in front of station 2 will be reduced.

A new control parameter called *delay_time* is introduced to the model to account for the difference in average processing times ΔPS. It should be noted that the presence of station breakdowns and randomness of the parameters make it very difficult to estimate the exact value of the delay parameter. If the process parameters were deterministic and there were no station break-downs, *delay_time* would simply be the difference between service times, and all the stations would operate with zero inventory. However, break-downs cause downstream stations to starve which reduces the efficiency and the profitability of the entire line. To prevent or reduce starvation, stations need to allow some inventory build-up in their buffers. By doing that they are be able to continue their operation even if their upstream counterparts are down and the material flow is temporarily cut-off. In the control strategy detailed in the

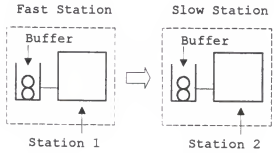


Figure 3-3. A segment of a production line; A fast (PS_1) station and a slow (PS_2) station.

previous section, this was accomplished by adjusting the buffer size. Here, the inventory levels are controlled by varying the *delay_time* parameter of each station. Clearly, a high *delay_time* will result in a low inventory and a low inventory cost. By the same token, stations operating with low *delay_time* values will experience higher inventory levels and high inventory costs.

The rule set for this control strategy is the same as the previous set with the exception of Rule 3 and Rule 4. It is as follows:

Rule 1. If a station is *blocked* because its downstream station does not grant permission to unload, then it does not authorize its upstream station to unload. The station remains *blocked*.

Rule 2. If a station is *down* and is being repaired, then it does not authorize its upstream station to unload. The station remains *down*.

Rule 3. If a station is *starving*, that is, functional but not processing any part at the moment because its buffer is empty, then it authorizes its upstream station to unload. The station becomes *up and running*.

Rule 4. If a station is *up* and *running*, that is, currently processing a part, then it waits for a duration of *delay_time* before it authorizes the upstream station to unload. The station remains *up and running*.

It should be noted that unlike the previous heuristic (Section 3.3.1) the buffer size is not a factor in the decision process, therefore, does not appear in the rule-base (Rule 3 and Rule 4). In other words, buffers with infinite capacity are assumed. To make it more applicable to real-life manufacturing problems a more realistic upper limit may be used. As stated later, a finite buffer size (9) is used in the simulations.

Another important point is that only Rule 4 contains the control parameter *delay_time*. The reasoning behind this is that if the station is starved, delaying the arrival of part will interrupt the production needlessly resulting in a loss of income. If the station is blocked or down, the part will block the upstream station and will be allowed into the buffer as soon as the station becomes starved.

This control strategy has been tested with the learning heuristics similar to those described in section 2.3.1. These heuristics are based on the manipulation of the control parameter *delay_time*. Both originate from the same idea; each station observes the trend in the profit made and adjusts the *delay_time* according to the observation. In rule format,

Rule 1. If at the beginning of the last interval *delay_time* had been increased, and as a result the *profit* went up, then increase *delay_time*.

Rule 2. If at the beginning of the last interval *delay_time* had been reduced, and as a result the *profit* went up, then reduce *delay_time*.

Rule 3. If at the beginning of the last interval *delay_time* had been increased, and as a result the *profit* went down, then reduce *delay_time*.

Rule 4. If at the beginning of the last interval *delay_time* had been reduced, and as a result the *profit* went down, then increase *delay_time*.

The refined version expands Rule 3 and Rule 4 as follows:

Rule 3a. If at the beginning of the last interval *delay_time* had been increased, and as a result the *profit* went down and a lower average inventory level is achieved, then reduce *delay_time*.

Rule 3b. If at the beginning of the last interval *delay_time* had been increased, and as a result the *profit* went down and a higher average inventory level is achieved, then increase *delay_time*.

Rule 4a. If at the beginning of the last interval *delay_time* had been reduced, and as a result the *profit* went down and less parts have been processed, then reduce *delay_time*.

Rule 4b. If at the beginning of the last interval *delay_time* had been reduced, and as a result the *profit* went down and more parts have been processed, then increase *delay_time*.

As stated before, the inventory level in a buffer is inversely proportional to *delay_time*. It is expected that

increasing the *delay_time* will reduce the inventory level and consequently the inventory cost.

CHAPTER 4
SIMULATIONS

4.1 Overview

Adaptive control policies *buffer_size* and *delay_time* have been described in great detail in Chapter 3. This chapter focuses on the analysis of their performance based on the simulations performed using various cases. For comparison purposes two static control policies have been selected: a very conservative pull policy (one-card kanban), and an aggressive pull policy (ten-card kanban). These have also been tested using the same setups and the parameter values. Moreover, extensive simulations have been performed to find the kanban policy with the optimum card allocation that yields the maximum profit in all cases.

Section 4.2 describes the selected static control policies while Section 4.3 concentrates on five different scenarios designed for the simulation of the production line and lists the parameter values used in the simulations. The simulation results for all the heuristics are presented and explained in Section 4.4. Section 4.5 deals with the evaluation of the simulation results whereas Section 4.6 is dedicated to the discussion of adaptive behavior in general and includes observations made during the analysis of results.

4.2 Static Control Policies

The static control policies selected operate based on the following principles:

Rule 1. If a station is *blocked,* it does not accept any new materials into its buffer.

Rule 2. If a station is *down*, it does not accept any new materials into its buffer.

Rule 3. If a station is *starved*, it accepts the new material and becomes *up and running.*

Rule 4. If a station is *up and running*, it accepts new material into its buffer if a card is available.

The rules stated above are similar to the rules constituting the rule base of the adaptive control policies *delay_time* and *buffer_size.* In ten-card kanban there are ten cards in each station, therefore there can be a maximum of nine parts waiting in the buffer at any time. In other words, the maximum buffer capacity is nine. In one-card kanban, on the other hand, only one card is available for each station, which means that, while processing a part, a station cannot accept another. Therefore, transfer of parts from one station to another will occur only if the second station is starved. That makes Rule 4 redundant for one-card kanban. It should be noted that because of the availability of only one card, the one-card kanban policy operates with zero inventory level at all times. Needless to say, the static control policies do not include any learning mechanism.

4.3 Simulation Scenarios

The adaptive control strategies (with learning heuristics) as well as the static control policies described in the previous sections have been tested in five different production line setups. Each setup was carefully designed to represent a different manufacturing scenario. The selected scenarios are:

(a) No bottleneck,

(b) One bottleneck (mild),

(c) One bottleneck (severe),

(d) Variable bottleneck,

(e) Two bottlenecks.

Each setup consists of four stations (S1, S2, S3, S4), each with an input buffer. A maximum buffer capacity of 9 has been used for all the station buffers except the first one, in the simulations. The first station (S1) has an infinite buffer connected to an infinite supply of parts. For this reason, the first station never starves and is in continuous production as long as it is not down and not blocked by S2.

As explained in Chapter 3, stations (except S1) control the influx of materials into their buffers. Every part in the input buffer of a station waiting to be processed costs an inventory penalty of $1 per unit time to that station. The only exception is the first station, which operates cost-free. A station is in one of the following four states at any time: UP and RUNNING (WORKING), STARVED, DOWN, and BLOCKED. If a station does not receive permission to unload the part it just finished

processing into the input buffer of the next station, it becomes BLOCKED.

In all five scenarios explained below, each station is characterized by the following parameters: service_time (process time), time_to_breakdown, and time_to_repair. Service time is the amount of time it takes for the station to process a part. Time_to_breakdown is the duration a station can be UP and RUNNING before it breaks down. Therefore, a station can break down only while it is UP and RUNNING. Time_to_repair is how long it takes to repair a station that is DOWN. These are random parameters, which are uniformly distributed; the first value is the lower limit, whereas the second value represents the upper limit.

4.3.1 Case 1: No Bottleneck (Balanced Line)

In this setup all four stations have identical parameter values:

[S1]-[S2]-[S3]-[S4] :

Service_time : 30 - 50 time units,

Time_to_breakdown : 600 - 1000 time units,

Time_to_repair : 160 - 240 time units.

On average each station remains in production for 800 time units, and in repair for 200 time units. Assuming that a station does not starve or gets blocked at all, it will be operational 80% of the time and down 20% of the time. By the same token, it takes a station, on average, 40 time units to process a part. Again assuming no starving and no blocking a station can service

20 parts in 1000 time units. Consequently, in a push system with infinite buffers the maximum expected production rate is 20 in 1000 time units.

4.3.2 Case 2: Single Mild Bottleneck

In this setup stations 1, 2 and 4 have identical parameter values, while station 3 has a higher service_time.

[S1]-[S2]-[S4] :

Service_time : 30 - 50 time units,

Time_to_breakdown : 600 - 1000 time units,

Time_to_repair : 160 - 240 time units.

[S3] :

Service_time : 40 - 60 time units,

Time_to_breakdown : 600 - 1000 time units,

Time_to_repair : 160 - 240 time units.

In this scenario, station 3 (S3) is the clear-cut bottleneck of the production line. Note that its production rate is less than that of the other three stations. Therefore, there will be a significant amount of accumulation of parts in the input buffer of S3. In a push system that inventory level will depend on the buffer capacity, whereas in a kanban system it will depend on the number of cards used.

Since a serial production line can work as fast as its slowest component, in this scenario the maximum expected production rate in a push system with infinite buffers is 16 in 1000 time units.

4.3.3 Case 3: Single Severe Bottleneck

This setup is similar to the previous setup in that it consists of three fast stations with identical parameter values and a slower station with a high service time. However, here the slow station is even a bigger bottleneck than the previous case since its service time is even higher.

[S1]-[S2]-[S4] :

Service_time : 30 - 50 time units,

Time_to_breakdown : 600 - 1000 time units,

Time_to_repair : 160 - 240 time units.

[S3] :

Service_time : 60 - 100 time units,

Time_to_breakdown : 600 - 1000 time units,

Time_to_repair : 160 - 240 time units.

Because of S3 the production in this setup is considerably slower than the previous two; the maximum expected production rate in a push system with infinite buffers is only 10 in 1000 time units.

4.3.4 Case 4: Variable Bottleneck

The production line in this setup consists of three different type of stations: S1 and S2 are identical fast stations with frequent breakdowns, S3 is a slow station with no breakdown and S4 is a fast station with no breakdown.

Even though S3 has a relatively high service time compared to S1 and S2, its stand-alone production rate is about the same because it is always operational whereas S1 and S2 are

[S1]-[S2] :

Service_time : 30 - 50 time units,

Time_to_breakdown : 160 - 240 time units,

Time_to_repair : 160 - 240 time units.

[S3] :

Service_time : 60 - 100 time units,

Time_to_breakdown : ∞ (infinite),

Time_to_repair : 0.

[S4] :

Service_time : 30 - 50 time units,

Time_to_breakdown : ∞ (infinite),

Time_to_repair : 0.

operational about 50% of the time. The input buffer of S3 is where the majority of inventory build-up is expected to occur in both the static and the adaptive control strategies. The excess input to S3, which accumulates in the buffer while S2 is operational, can be consumed by S3 during the down period of S2. How big of a bottleneck S3 will be depends on how much the line can tolerate the accumulation of parts in S3's input buffer. For systems operating with high unit inventory costs high inventory levels in front of S3 will have a negative effect on the profit. As the unit inventory cost goes down (or the production reward goes up) the line will be able to tolerate a higher level of inventory in S3's buffer and consequently higher throughput and profit will be achieved with a higher buffer capacity (or kanban cards).

On the other hand, the last station (S4) has almost no effect on the performance of the production line. It is twice as fast as S3 and does not break down either. For that reason, it is expected to act like an all-pass filter, that is, parts should go through S4 with minimum delay and therefore, only a minimal inventory build-up in front of S4 is expected.

In a push system with infinite buffers the maximum expected production rate is 12.5 per 1000 time units.

4.3.5 Case 5: Two Bottlenecks

This setup consists of three identical fast stations (S1, S2 and S4) with frequent breakdowns and one slower station with no breakdown. It differs from the previous case in that the last station (S4) does break down which might create another bottleneck in the system.

[S1]-[S2]-[S4] :

Service_time : 30 - 50 time units,

Time_to_breakdown : 160 - 240 time units,

Time_to_repair : 160 - 240 time units.

[S3] :

Service_time : 60 - 100 time units,

Time_to_breakdown : ∞ (infinite),

Time_to_repair : 0.

The fact that S4 breaks down rather frequently suggests the possibility that it also might interrupt the material flow in the system and therefore become a second bottleneck. The performance of the line is not only determined by the inventory

accumulation in S3's buffer and by the rate S3 processes parts, but also by how big of a hindrance S4 will be to the flow of parts. In case 4 explained earlier, S4 does not block S3 at all because it is faster than S3 and operates without breaking down. However, in this setup S4 is down for a considerable amount of time (approximately 50% of the time) and may block S3 significantly.

4.3.6 Issues Involving Simulation Parameters

Runs have been performed for three different reward-cost ratios in each setup:

(a) Production Reward = $500 per unit
    Inventory Cost = $1 per part per unit time

(b) Production Reward = $1000 per unit
    Inventory Cost = $1 per part per unit time

(c) Production Reward = $1500 per unit
    Inventory Cost = $1 per part per unit time

An important issue here is the distribution of the production reward to individual stations. One approach is to divide it evenly and assign one fourth of the overall reward to each station. Another approach is to base it on the stand-alone production rates. The stand-alone production rate R_{SA} of a station can be calculated by

$$R_{SA} = \frac{T_{BD}}{T_S(T_{BD} + T_R)} \qquad (4\text{-}1)$$

where T_{BD} is the time_to_breakdown, T_R is the time_to_repair and T_S is the service_time. $1/R_{SA}$ gives the average time it takes a station to process a part assuming no starvation or blocking.

This method encourages the potential bottlenecks (stations with low production rates) of the system to be more active and consequently increases the throughput. At the same time it increases the inventory level in front of the bottleneck station and drives the inventory cost up.

In this study a third approach is used to distribute the production reward among the stations. In the simulations presented here, the reward a station receives for producing a part is proportional to the service time of that station. Hence, fast stations receive less reward per part than slow stations.

In the simulation of each case a run length of one million time units is used with an interval length of ten thousand time units. The determination of interval length as well as the number of intervals is an important issue for adaptive control strategies.

Intervals should be long enough so that the system can settle down and has an opportunity to correctly evaluate the effects of the changes made by the learning module at the end of each interval. If the interval is not long enough for the transients to die out, the transients will influence the decision making process more than the steady-state behavior. Another advantage of long intervals is that it minimizes the effects of randomness in the system. Due to the non-deterministic nature of the processes (represented by the uniformly distributed service, breakdown and repair times in this study) the performance of the production line might vary significantly in short term under the same or similar conditions. This instability of the performance makes it

difficult for the learning module to make the right decision. This is especially true for *delay_time* heuristic where the tuning of parameter *delay_time* is done in small incremental steps. The solution to this problem is obviously to observe the system for a long period of time so that the random fluctuations in process times even out.

On the other hand, if the interval length is too long and consequently there is only a small number of intervals, the adaptiveness of the system will suffer. The system will require a longer time to adjust when drastic setup changes occur, that is, it will learn slower.

Therefore, there is a compromise between the stability and the adaptiveness (agility) of the system.

During the run, the learning module continuously adjusts the value of the control parameter of each station to find a more profitable operating point. After every interval the control parameter is moved up or down by the pre-determined step size. The value of the step size determines how quickly the system can adapt to its environment. For large step sizes the system will be able to adapt to even the drastic changes of the operating conditions rather quickly but will have a hard time to fine-tune. On the other hand systems using small step sizes will take a long time to adjust themselves if significant changes in the manufacturing process occur, however, once they get close to the optimum operating point they will do a better job in staying close to that optimum operating point. For a more detailed discussion on step size the reader is referred to Section 4.6.2.

For adaptive control strategy *buffer_size* a step size of one is used in all the simulations. The buffer capacity therefore varies between 0 and 9 during the simulation.

For *delay_time* a step size of two time units is used in all the simulations. The wait period for a part, that is, the amount of time the part waits before being accepted to the next buffer, may vary between 0 and 200 time units.

4.4 Simulations

Simulation results for all the cases described above are presented in this section in tabular format and using plots. Adaptive control strategies *delay_time* and *buffer_size* have been tested in each setup for three different reward-cost ratios with pre-learn. In pre-learn mode an initial run is performed to determine an appropriate initial value for the decision parameter (*delay_time* or *buffer_size*). A single run of five million time units with interval length of ten thousand time units provides an educated guess for the proper initial value of the decision parameter. The idea behind this is to speed up the learning process by starting at a good operating point and thus to enable the system to reach steady-state quickly.

In real world applications, this pre-run can be performed by the stations virtually. Intelligent stations can be aware of their process parameters and use them in the pre-run. The production line can emulate the whole manufacturing process without actually producing a single product. The stations in the network can send messages to each other as if they are in real

production. Since it does not involve the processing of actual parts, the process of pre-learning can be completed in a matter of minutes.

4.4.1 Case 1: No Bottleneck (Balanced Line)

Two adaptive and two static control policies have been simulated for three different reward-cost ratios. Also numerous simulations have been performed to find out the best card combination (best case kanban) for each reward-cost ratio to serve as a reference point.

A pre-run of a duration of five million time units is performed for each adaptive strategy. The average value for the control parameter during the pre-run is used for the starting value of that parameter in the actual run. Note that the control parameter for S1 is not listed in the tables below. This is due to the fact that S1 is the first station in the production line and does not interrupt the material flow at all.

For control strategy *buffer_size,* the pre-run yielded the values listed in Table 4-1. As the table shows, as the reward-cost ratio goes up, the production line is encouraged to produce more parts, hence the buffer capacity increases.

For control strategy *delay_time,* the pre-run yielded the values listed in Table 4-2. The table indicates that with increasing reward-cost ratios the average wait time for parts decreases, which causes more parts to be produced.

The profits obtained from the simulation of all four control strategies and the best kanban policies are listed in Table 4-3. The values in parenthesis represent the number of

Table 4-1. Control parameter *buffer size* after pre-run (Case 1).

Buffer_size	**S2**	**S3**	**S4**
Reward = $500	0	1	8
Reward = $1000	1	5	8
Reward = $1500	4	7	8

Table 4-2. Control parameter *delay time* after pre-run (Case 1).

Delay_time	**S2**	**S3**	**S4**
Reward = $500	13	6	3
Reward = $1000	9	5	1
Reward = $1500	6	2	1

units produced in each control strategy. Note that each row in the last column corresponds to a different card combination (a different kanban).

Simulations reveal that the optimum card allocation for reward $500 is (1, 3, >4). The first number indicates the number of cards assigned to S2, second number represents the number of cards assigned to S3 and the third number is the number of cards assigned to S4. The > sign indicates that the profit remains the same even if more than 4 cards are allocated for S4. In other

Table 4-3. Profit (in thousand dollars) and throughput as a function of reward (Case 1).

Reward	One-card Kanban (K_1)	Ten-card Kanban (K_10)	Buffer_size w/Pre-learn (B_L)	Delay_time w/Pre-Learn (D_L)	Best Case Kanban
$500	$6,287 (12,574)	$5,516 (14,697)	$6,198 (13,260)	$6,231 (13,178)	$6,366 (13,400)
$1000	$12,574 (12,574)	$12,856 (14,697)	$13,094 (14,226)	$12,846 (13,422)	$13,152 (14,318)
$1500	$18,861 (12,574)	$20,213 (14,697)	$20,241 (14,299)	$19,860 (13,975)	$20,399 (14,632)

words, (1, 3, 4) yields the same profit as (1, 3, 10). It should be noted that

$$\text{Buffer\_Size} = \#\_\text{of\_Kanban\_cards} - 1 \qquad (4\text{-}2)$$

when comparing the kanban policy to *buffer_size*. For reward $1000 the best card combination turns out to be (2, 6, >7) whereas for reward $1500 (4, 6, >6) yields the maximum profit. The comparison of the best kanban policies to the adaptive policy *buffer_size* shows that the number of kanban cards are in agreement with the *buffer_size* values of the adaptive control policy.

As expected, the static control policies do not have the ability to adjust the way they operate which is evidenced by the fixed production count for different reward values. On the other hand, the adaptive policies increase their throughput as the production of parts becomes more profitable. They accomplish

this by either increasing the capacity of their buffer (Table 4-1) or by reducing the wait period (Table 4-2) for incoming parts. Figure 4-1 shows how the production in strategies using adaptive control goes up with increasing reward-cost ratio. This jump in production count is achieved at the expense of higher average inventory levels.

Figure 4-1 reveals that *buffer_size* responds to the increase in reward-cost ratio (R/C) first (R/C 500 → 1000) very aggressively by producing an additional 1000 units, but then the production count levels off and remains almost constant (R/C 1000 → 1500). This is because the production line reaches its feasible production limit at around R/C=1000; after that point the production of additional units can only be achieved at a much greater inventory cost.

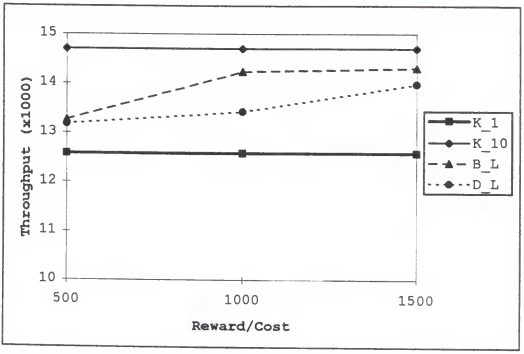


Figure 4-1. Variation of throughput in Case 1.

The other adaptive algorithm *delay_time* displays a somewhat different behavior. Between R/C 500 and 1000, the production count increases slightly, however in the range of 1000 to 1500 it makes a significant jump.

Ten-card kanban represents one extreme where all the stations operate with ten cards. Therefore, it is the upper limit for both adaptive control policies in terms of throughput. A comparison of the production counts indicate that even at R/C=1500, *buffer_size* and *delay_time* produce approximately 400 and 700 less parts respectively, showing that the production of those last 400 units comes at such a cost that it is not feasible.

To get a better idea of how the variation of reward affects the performance in each control policy, the profit percentages are plotted with respect to the reward-cost ratio. In Figure 4-2, 100% profit indicates the maximum profit by any control policy.

As the plot shows, the kanban policies with optimum card allocations yield the most profit for all reward-cost ratios. For a low reward-cost ratio (500 to 600) the conservative one-card kanban policy is closest to the best. The adaptive policies yield profits within 3% of that of the maximum profit. Ten-card kanban, due to high levels of inventory, works very inefficiently and turns in profits as low as 87% of maximum profit (at R/C=500).

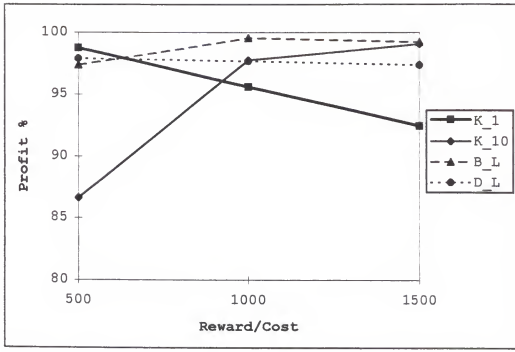


Figure 4-2. Variation of profit in Case 1.

As the reward-cost ratio goes up, ten-card kanban approaches one-card kanban until it catches up at about R/C=900. In this range the adaptive policies appear to be operating more efficiently.

After R/C=1000, ten-card kanban improves steadily while the *buffer_size* is still the most profitable with *delay_time* not far behind. One-card kanban falls behind generating only 92% of maximum profit (at R/C=1500).

It can be concluded that overall the adaptive policies perform better than the two extreme case static policies, *buffer_size* being the most successful policy. Both adaptive policies perform within 3% of the best kanban policy at each reward level.

4.4.2 Case 2: Single Mild Bottleneck

Like the previous case, the simulation of the production line is done for each control policy for a duration of one million time units. For the adaptive strategies, a pre-run of five million time units is performed to determine the starting parameter values. The parameter values used for the actual runs are given in Tables 4-4 and 4-5.

This is a case with a clear bottleneck (station 3). As the results in Table 4-6 show, the conservative method (one-card kanban) performs much better than the aggressive method (ten-card kanban). In ten-card kanban, the input buffer of S2 and especially S3 saturate which leads to a high inventory cost and

Table 4-4. Control parameter *buffer_size* after pre-run (Case 2).

Buffer_size	**S2**	**S3**	**S4**
Reward = $500	0	0	9
Reward = $1000	0	0	9
Reward = $1500	1	1	9

Table 4-5. Control parameter *delay_time* after pre-run (Case 2).

Delay_time	**S2**	**S3**	**S4**
Reward = $500	16	19	1
Reward = $1000	13	12	1
Reward = $1500	13	11	1

inefficient operation of the production line. The high inventory levels cause the system to even lose money (negative profit) for low reward-cost ratios.

The adaptive policies respond to reward variations similarly. Only very high reward-cost ratios (above 1000) promote a production increase. Even then, the *buffer_size* and *delay_time* values do not change much, resulting in only a limited amount of increase in throughput. As Figure 4-3 indicates, the adaptive strategies produce significantly less than the maximum which is throughput of ten-card kanban.

The card combination (1, 1, >1) represents the best kanban policy for reward $500. When reward goes up to $1000 the best card combination turns out to be (1, 2, >3). For reward $1500 (2, 3, >3) yields the maximum profit. Keeping in mind that

Buffer_Size = #_of_Kanban_cards - 1

Table 4-6. Profit (in thousand dollars) and throughput as a function of reward (Case 2).

Reward	One-card Kanban (K_1)	Ten-card Kanban (K_10)	Buffer_size W/Pre-learn (B_L)	Delay_time w/Pre-Learn (D_L)	Best Case Kanban
$500	$5,732 (11,464)	-$2,020 (14,240)	$5,579 (11,932)	$5,735 (11,972)	$5,807 (11,645)
$1000	$11,464 (11,464)	$5,100 (14,240)	$11,537 (12,047)	$11,675 (12,054)	$11,799 (12,448)
$1500	$17,196 (11,464)	$12,220 (14,240)	$17,364 (12,235)	$17,775 (12,356)	$18,037 (13,319)

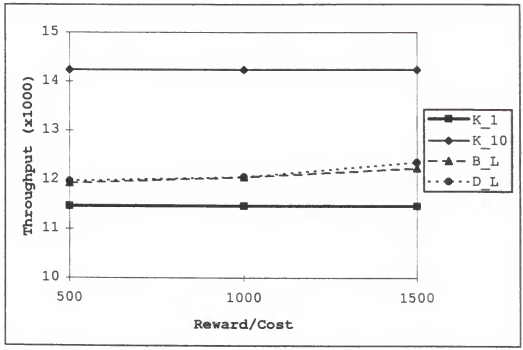


Figure 4-3. Variation of throughput in Case 2.

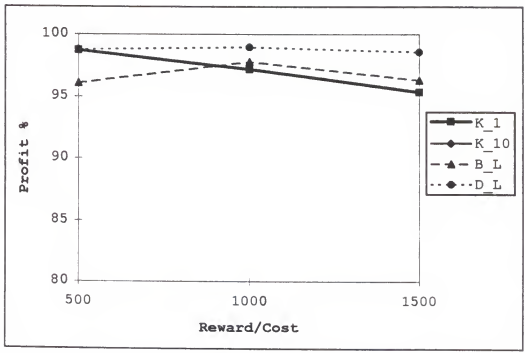


Figure 4-4. Variation of profit in Case 2.

one concludes that the adaptive policy *buffer_size* behaves similar to the best kanban policy by limiting its buffer capacity. Figure 4-4, which is the plot of profit percentages with respect to reward-cost ratios, shows that the conservative one-card kanban policy and the adaptive policies yield an overall high profit whereas the ten-card kanban policy fails catastrophically. In fact, ten-card kanban performs so poorly that it does not appear in the plot. One-card kanban becomes less profitable as the reward goes up. The adaptive policies respond to increasing reward by increasing the buffer capacity (*buffer_size*) or reducing the wait period (*delay_time*) slightly.

One possible explanation for the superior performance of *delay_time* over *buffer_size* is that the small step size used in the *delay_time* algorithm allows a better fine-tuning of the control parameter. The learning algorithm constantly searches for a better operating point by varying the control parameter. In *delay_time*, the control parameter is the wait period which in every interval is moved up or down by 2, a rather small step size. In *buffer_size* however the buffer capacity, the control parameter, is either increased or decreased by 1 which is the minimum possible step size. Analysis shows that in this particular case the performance of a production system changes significantly when the buffer capacity goes from 0 to 1. Therefore, changing *buffer_size* by 1 has a larger effect on the profit than changing *delay_time* by 2. Due to this inherent handicap the *buffer_size* algorithm yields a slightly lower profit.

4.4.3 Case 3: Single Severe Bottleneck

This setup is an exaggerated version of the setup in the previous case. Just like case 2, the production line consists of three identical fast stations (S1, S2 and S4) and a slow station (S3). The difference is that here S3 is even a bigger bottleneck. It works twice as slow as the other stations but it is operational for the same amount of time on average. In a push system this would lead to a large amount of accumulation in front of S3.

The values of the control parameters determined by pre-runs are listed below.

Table 4-7. Control parameter *buffer_size* after pre-run (Case 3).

Buffer_size	**S2**	**S3**	**S4**
Reward = $500	0	0	8
Reward = $1000	0	0	8
Reward = $1500	0	0	8

Table 4-8. Control parameter *delay_time* after pre-run (Case 3).

delay_time	**S2**	**S3**	**S4**
Reward = $500	25	50	3
Reward = $1000	23	43	2
Reward = $1500	21	41	2

The tables indicate that adaptive policies *buffer_size* and *delay_time* follow a very conservative strategy. *Buffer_size* operates very conservatively with zero buffer capacity, thus mimics one-card kanban. Stations 2 and 3 accept parts only if they are starved. The bottleneck station S3, which processes parts at a much lower stand-alone rate than S2, operates with zero buffer capacity to avoid high inventory costs which causes S2 to be blocked most of the time. To prevent loss of profit S2 synchronizes itself with S3 by reducing its buffer capacity to zero as well.

A look at Table 4-8 reveals a similar situation for *delay_time*. The bottleneck station S3 imposes a long wait on every incoming part (50 time units for R/C=500) because parts arrive faster than S3 can process them. This interrupts the material flow and blocks the output of S2 to which S2 responds by restricting the input to its buffer with a delay of its own.

One interesting observation is that, unlike the previous cases, the reward has almost no effect on the control parameters and consequently the throughput (Figure 4-5). The tables show that the buffer capacity does not vary with reward at all while wait period decreases slightly. Compared to case 2, the overall dependence on reward is weaker. This is most probably due to the extreme difference between the stand-alone production rates of S3 and the other stations. Even a slight change in control parameters results in a high inventory cost for S3. Even very high reward values cannot convince S3 to increase its buffer capacity or reduce the wait time. This means that S3 is already

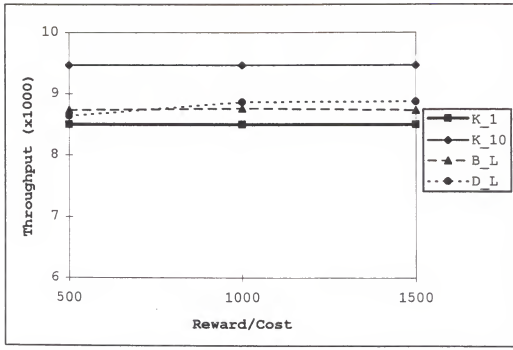


Figure 4-5. Variation of throughput in Case 3.

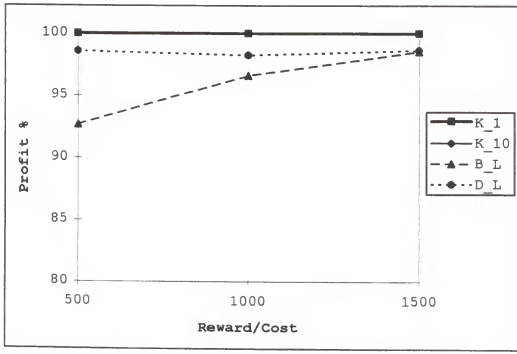


Figure 4-6. Variation of profit in Case 3.

Table 4-9. Profit (in thousand dollars) and throughput as a function of reward (Case 3).

Reward	One-card Kanban (K_1)	Ten-card Kanban (K_10)	Buffer_size w/Pre-learn (B_L)	Delay_time w/Pre-Learn (D_L)	Best Case Kanban
$500	$4,249 (8,498)	-$12,602 (9,470)	$3,937 (8,732)	$4,189 (8,635)	$4,249 (8,498)
$1000	$8,498 (8,498)	-$7,867 (9,470)	$8,208 (8,757)	$8,350 (8,865)	$8,498 (8,498)
$1500	$12,747 (8,498)	-$3,132 (9,470)	$12,559 (8,734)	$12,763 (8,873)	$12,747 (8,498)

near its production limit; a higher buffer capacity or a lower wait time will directly translate into a higher inventory with minimal additional production. It should be noted that due to its design, *delay_time* is better at fine-tuning, that is, it can track the desired operating point better. Owing to its small step size, it is more responsive to the changes in reward as evidenced by Figures 4-5 and 4-6.

The comparison of profits in Table 4-9 clearly shows that one-card kanban is the best in this case among all policies. According to simulations conducted using various card combinations, (1, 1, >1) yields the best results for all the reward values which is in agreement with the adaptive control policy *buffer_size*. Ten-card kanban is easily the worst performer yielding negative profit even for R/C=1500.

Both adaptive control strategies are within 2% of one-card kanban at the high end, however as the reward goes down *buffer_size* starts falling behind. At R/C=500 it can generate only 93% of the maximum profit. As discussed above this is a result of systematic variations in buffer capacity by the learning algorithm to find a better operating point. Even though its control parameter experiences the same kind of variations, *delay_time* does a better job of staying close to one-card kanban because of its ability to increase and decrease its control parameter in small amounts.

4.4.4 Case 4: Variable Bottleneck

The initial simulation of the setup described in Section 4.3.4 for five million time units provides the following values for the control parameters.

Tables 4-10 and 4-11 clearly show the strong dependency of the control parameters on reward. The changes in reward result in considerable jumps in *buffer_size* and *delay_time*. A careful look at these tables as well as Figure 4-7 reveals the similar behavior patterns displayed by the adaptive control strategies.

Table 4-10. Control parameter *buffer_size* after pre-run (Case 4).

Buffer_size	**S2**	**S3**	**S4**
Reward = $500	1	1	9
Reward = $1000	4	7	9
Reward = $1500	7	8	9

Table 4-11. Control parameter *delay_time* after pre-run (Case 4).

Delay_time	**S2**	**S3**	**S4**
Reward = $500	14	12	0
Reward = $1000	6	3	0
Reward = $1500	2	1	0

For R/C=500, S2 and S3 prefer relatively low inventory levels, which they accomplish by keeping the buffer capacity at 1 (*buffer_size*) and administering rather long delays of 14 and 12 time units (*delay_time*), respectively. Under these conditions both control strategies operate very efficiently; their profit is high (Figure 4-8) even though their throughput is much less than that of ten-card kanban (Figure 4-7).

When R/C goes from 500 up to 1000, S2 and S3 react by increasing the average inventory they keep noticeably. They do this by raising their buffer capacity to 4 and 7, and reducing the delay to 6 and 3 time units, respectively. As a result, the throughput of both adaptive strategies increase significantly while their profit, in spite of high inventory costs, is still within 2% of ten-card kanban.

Finally, at the highest reward level *buffer_size* and *delay_time* operate almost in push fashion, that is, at near maximum buffer capacity (7 and 8) and with negligible delays (2 and 1 time units). Consequently, their inventory levels, throughput and profit are very similar to those of ten-card kanban.

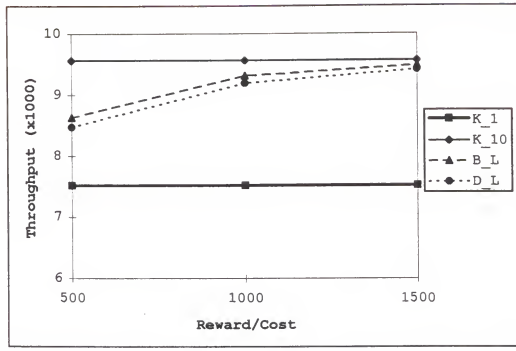


Figure 4-7. Variation of throughput in Case 4.

As predicted in Section 4.3.4, S4 does not impose any restrictions to the material flow at all. Since it is up all the time and has a much higher processing speed than that of S3 it operates virtually inventory-free even with maximum buffer capacity and zero delay.

As discussed in Section 4.3.4, S3 is a potential bottleneck in this setup due to its low processing speed. The throughput of the whole line is determined by how much inventory S3 allows to accumulate in its buffer. If S3 decides to have a low buffer capacity or to impose large delays to incoming parts, the production will be limited. If, on the other hand, S3 finds it profitable to operate with large inventory levels the upstream

Table 4-12. Profit (in thousand dollars) and throughput as a function of reward (Case 4).

Reward	One-card Kanban (K_1)	Ten-card Kanban (K_10)	Buffer_size w/Pre-learn (B_L)	Delay_time w/Pre-Learn (D_L)	Best Case Kanban
$500	$3,759 (7,517)	$3,890 (9,564)	$3,874 (8,628)	$3,896 (8,471)	$3,950 (9,473)
$1000	$7,517 (7,517)	$8,672 (9,564)	$8,553 (9,314)	$8,504 (9,191)	$8,672 (9,582)
$1500	$11,276 (7,517)	$13,454 (9,564)	$13,377 (9,483)	$13,330 (9,413)	$13,454 (9,582)

stations will be blocked less and the throughput will approach ten-card kanban limit.

From the inspection of the Figure 4-7 it is evident that for low R/C the former is true; S3 becomes the bottleneck of the system in terms of production. Yet it produces enough to keep the system very profitable. For high R/C ratios however, the latter is the case; S3 does not hinder production.

Additional simulations show that the best kanban cases correspond to the card combinations (2, 3, >3) for reward $500, (>5, >7, >7) for reward $1000 and (>5, >7, >7) for reward $1500. For rewards $1000 and $1500, increasing the number of cards above (5, 7, 7) does not have any effect on the profit at all. That is why ten-card kanban (10, 10, 10) yields the same profit as the best kanban in Table 4-12. Again the optimum card allocation in kanban agrees with the *buffer_size* values of the adaptive control policy quite well.

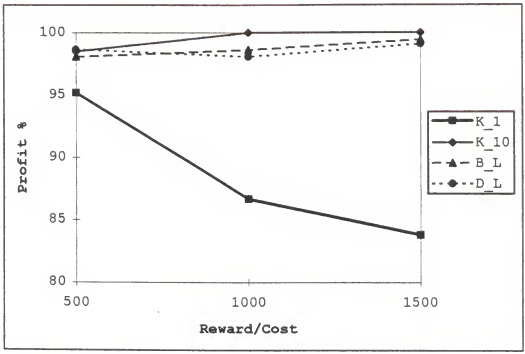


Figure 4-8. Variation of profit in Case 4.

As far as the overall performance is concerned, this is a case where the aggressive policy outperforms the conservative policy especially for high R/C values. Adaptive policies seem to be doing a good job in the whole range of R/C values, much better than one-card kanban which performs progressively worse as R/C increases.

4.4.5 Case 5: Two Bottlenecks

This is a more complicated version of case 4. The last station spends a considerable amount of time being down which adds a new dimension to the problem. As a result of frequent breakdowns to S4, the output of S3 gets blocked interrupting the material flow from S3 to S4. A direct result of this is the reduced throughput as shown in Figure 4-9.

In this setup, one-card kanban suffers from the interrupted flow caused by S3's inability to store the excess input parts in its buffer so that it can process them later when S2 is down. This is detrimental to the profitability of the line especially for high rewards. On the contrary, ten-card kanban suffers from excessive inventory accumulation caused by the availability of too many cards. S4 simply does not let S3 to pass parts through by blocking its output.

Pre-runs of adaptive control strategies *buffer_size* and *delay_time* provide the following suggestions for control parameters (Table 4-13 and Table 4-14).

Table 4-13. Control parameter *buffer_size* after pre-run (Case 5).

Buffer_size	**S2**	**S3**	**S4**
Reward = $500	1	0	9
Reward = $1000	1	1	9
Reward = $1500	2	2	9

Table 4-14. Control parameter *delay_time* after pre-run (Case 5).

Delay_time	**S2**	**S3**	**S4**
Reward = $500	20	39	2
Reward = $1000	18	25	1
Reward = $1500	10	18	1

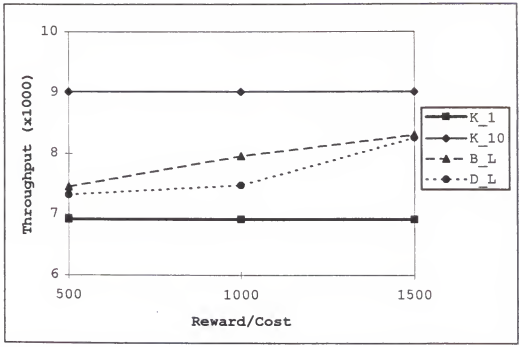


Figure 4-9. Variation of throughput in Case 5.

Compared to case 4, the production line operates with less buffer capacity (*buffer_size*) and higher delays (*delay_time*). In case 4, the breakdownless operation of S4 allows the slow station S3 to let parts accumulate in its buffer. When S2 goes down S3 is able to continue processing parts and pass them along to S4 since S4 does not block S3 at all. This mode of operation boosts up the production. In this setup however, S4 blocks the output of S3 not allowing it to process parts at the same rate as S2. Therefore, S3 has to limit its part intake to avoid a saturated inventory and does this by using a small buffer (*buffer_size*) and higher wait times (*delay_time*).

Increase in reward naturally promotes production, but because of S4's breakdowns the production count in both adaptive

policies (Figure 4-9) do not come close to that of ten-card kanban unlike case 4 (Figure 4-7).

Case 5 proves to be the most complicated case of all. None of the four policies can stay close to the best possible kanban policy for all R/C values. Nevertheless, the adaptive policies perform better than the static policies as evidenced by their relatively high profit percentages in Figure 4-10. Especially *delay_time* does a good job and easily outperforms all the others in almost the entire R/C range. It too however falls 5% behind the best kanban policy for R/C=1000. Just like case 4, one-card kanban does poorly for high R/C but to a lesser extent. As mentioned before, the availability of only one card in one-card

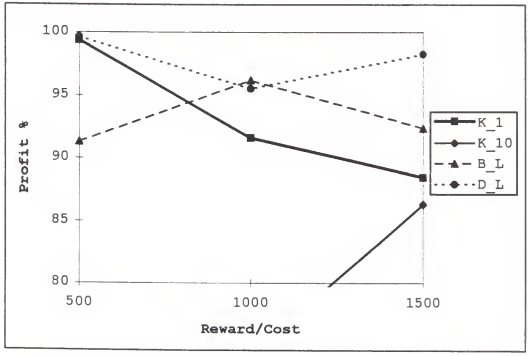


Figure 4-10. Variation of profit in Case 5.

Table 4-15. Profit(in thousand dollars) and throughput as a function of reward (Case 5).

Reward	One-card Kanban (K_1)	Ten-card Kanban (K_10)	Buffer_size w/Pre-learn (B_L)	Delay_time w/Pre-Learn (D_L)	Best Case Kanban
$500	$3,459 (6,918)	$1,106 (9,020)	$3,178 (7,445)	$3,468 (7,320)	$3,481 (7,911)
$1000	$6,918 (6,918)	$5,616 (9,020)	$7,265 (7,956)	$7,213 (7,473)	$7,553 (8,283)
$1500	$10,377 (6,918)	$10,126 (9,020)	$10,834 (8,306)	$11,527 (8,259)	$11,732 (8,701)

kanban makes it impossible for S3 to load up on parts and process them later when S2 is down. The negative effects of this however is not as serious as in case 4 because S3 gets blocked by S4 and cannot process as many parts anyway.

There is a substantial difference in performance for ten-card kanban between cases 4 and 5. The fact that S4 breaks down affects this strategy the most. S3 cannot process parts fast enough to keep up with S2 even though they have the same stand-alone production rates. That again is because S3 gets blocked by S4 and has to stop processing.

The inspection of Table 4-15 reveals that stations utilize their inventory much more efficiently when employing the adaptive strategy *delay_time* rather than *buffer_size*. Even though the throughput is less, the production line makes significantly more profit which can be attributed to the ability of *delay_time* to better fine-tune the behavior of stations.

Simulations performed to determine the best kanban card allocations reveal that for reward $500 (1, 2, 10) is the optimum card combination, whereas for reward $1000 and $1500 the optimum card combinations are (1, 3, 10) and (2, 3, 10), respectively.

4.4.6 Mixed Case

In order to demonstrate the ability of control strategies *buffer_size* and *delay_time* to adapt to changing setup conditions, a mixed case made up from four different scenarios is prepared. In this mixed run the production line consists of four stations as before. Production starts with the parameter values listed in case 1 (Section 4.3.1). However, at specific points during the run setup parameters are changed to observe how the system reacts to the changes made. After a duration of one million time units in setup 1, the production line switches to setup 2 (Section 4.3.2). After another one million time units the parameters change again, this time to setup 4 (Section 4.3.4). Finally, in the last one million time units the production line operates with the parameters of case 5 listed in Section 4.3.5. Throughout the run reward remains constant at $1000 while a $1 inventory cost is used per part per unit time.

At each transition, the stations in the line perform a pre-run to determine a new estimate for their control parameter. In a real manufacturing environment, the transitions from one case to another correspond to a station being modified, added to and/or taken out of the line. The nodes (stations) in the network are in constant communication with each other; any

station joining the network notifies the others of its existence. Similarly, each station broadcasts a message in periodic intervals to the entire network to report its current condition. This makes sure that all the nodes of the network are aware of any setup changes and perform a virtual run (pre-run) to prepare themselves.

In order to better demonstrate how the whole system adapts after every setup change, a mixed run without pre-run is performed as well. The plot in Figure 4-11 shows the profit over the course of the entire manufacturing session for the control strategy *buffer_size*. A sudden drop in profit observed at the first transition is due to the increase in bottleneck station S3's service time. As shown in Figure 4-12, S3 adjust to this by reducing its buffer capacity and eliminating the excess inventory. The next setup change does not appear to have any effect on the performance of the line. S3 is able to increase its buffer capacity right away taking advantage of breakdown-less operation. The observed fluctuations in the last quarter hint to some sort of instability in the system. One explanation is that the stations cannot adjust properly and overreact by reducing their buffer capacity too much. The system simply does not have enough time to recover and stabilize itself.

It should also be noted that even though it takes S3 almost the half of the first interval to raise its buffer capacity to a stable value the overall profit does not seem to suffer.

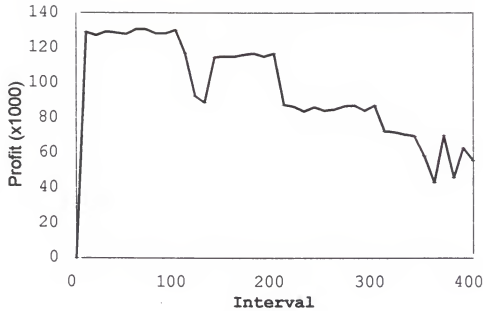


Figure 4-11. Variation of profit with time for *buffer_size* (no Pre-Run, R/C=1000).

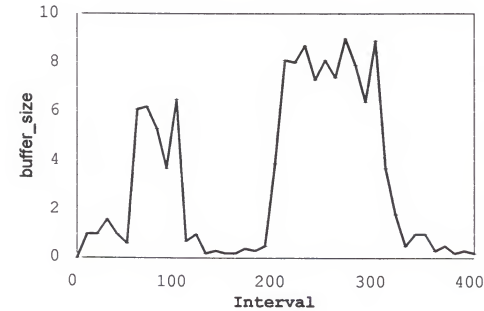


Figure 4-12. Variation of S3's control parameter with time (no Pre-Run, R/C=1000).

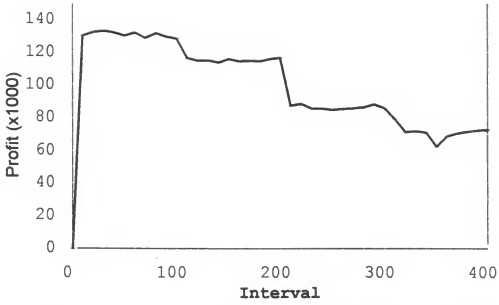


Figure 4-13. Variation of profit with time for *buffer_size* (Pre-Run, R/C=1000).

Additional simulations indicate that the increase in profit is insignificant for *buffer_size* above 1. S3's buffer capacity reaches 1 rather quickly and therefore no recovery period is needed. Pre-run allows much smoother transitions as shown in Figure 4-13. The system adjusts to the setup changes immediately which results in a much higher profit. In fact, the profit is within 4% of the combination of best kanban policies for all the reward-cost ratios (Figure 4-17).

When the same run is repeated for the other adaptive control strategy, *delay_time*, the production line appears to do better during the transition periods with no pre-run. In Figure 4-14, fluctuations in profit is observed following the first setup change at interval 100. The oscillations get smaller as

time goes by and finally die out at around interval 160. The comparison of Figure 4-14 and Figure 4-15 clearly shows the effect of the delay imposed by S3 on the profit. The bottleneck station S3 responds to the setup change by slowly increasing its delay parameter. The adjustments made by other stations, particularly S2 have an impact on profit too and cause the oscillatory behavior of the profit. When the delay parameter of S3 reaches the desired level the profit stabilizes and remains almost constant until the next setup change.

With the start of the third quarter (case 4) of the mixed run, the production line experiences a rather small drop in profit which can be attributed to high initial delay values. The conditions in case 4 dictate S3 to operate with a small delay to be profitable. However, due to the small step size used by *delay_time* an instantaneous reduction of the delay parameter is not possible. It takes a while to rid the system of the residual effects of the previous interval. During this period the profit increases slowly but steadily before finally reaching a plateau at interval 240.

S3 adjusts to the transition from case 4 to case 5 by immediately increasing its delay from nearly zero up to 30. This immediate response of S3 coupled with an even faster response of S2 which quickly increases its own delay parameter allows this transition to occur rather smoothly. As a side note, further simulations show that S3 overreacts by letting its delay parameter go up higher than desired.

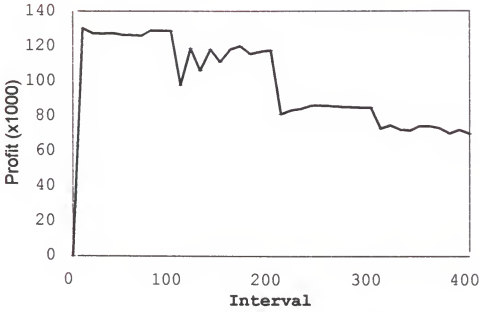


Figure 4-14. Variation of profit with time for *delay_time* (no Pre-Run, R/C=1000)

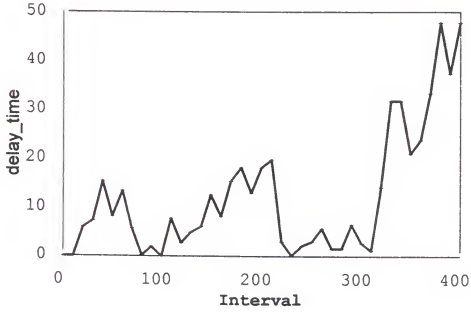


Figure 4-15. Variation of S3's control parameter with time (no Pre-Run, R/C=1000).

The delay patterns of other stations are not shown in Figure 4-15 to avoid overlapping plots and also because S3 is the bottleneck station in most cases and deserves the most attention.

The information obtained from the pre-run enables the production line to handle the transitions more easily and consequently, the resulting production generates more profit. From Figure 4-16 the only time the system struggles is in the second quarter of the mixed run (case 2) when it experiences fluctuations in profit similar to those of the previous run but to a lesser extent. During the remainder of the run the line is able to maintain a stable profit.

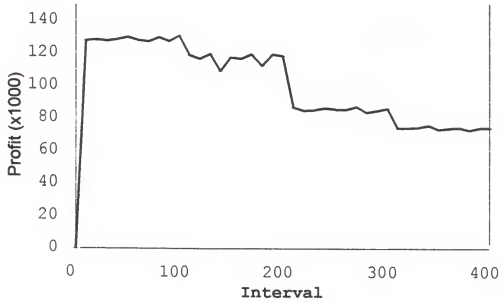


Figure 4-16. Variation of profit with time for *delay_time* (Pre-Run, R/C=1000).

The adaptive strategy *delay_time* proves to be profitable on a consistent basis. Just like *buffer_size* it yields profits only 2% - 4% less than the combination of best kanban policies over the entire spectrum of R/C values.

The adaptive policies *buffer_size* and *delay_time* display a similar profit pattern. For low R/C, *delay_time* is slightly better than *buffer_size*, for high R/C the roles are reversed. From Figure 4-17, both are superior to the conservative pull (one-card kanban) and aggressive pull (ten-card kanban) policies. As expected, one-card kanban loses its edge as R/C goes up. Ten-card kanban performs very poorly for low R/C mostly due to its ineffectiveness in case 3. Both static policies suffer for a certain range of R/C due to their inability to adjust.

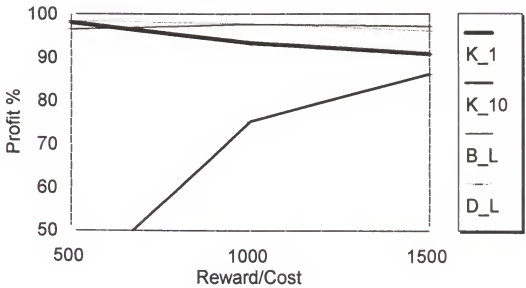


Figure 4-17. Variations in profit in the mixed case.

Overall, the mixed case proves to be very useful in determining how flexible and consistent the adaptive strategies are. It helps to demonstrate that they are able to learn and adapt, and that they perform consistently in a wide variety of manufacturing conditions for a large range of R/C values.

4.5 Evaluation of Simulation Results

In the previous section, the simulation results for two adaptive and two static control strategies have been presented. Additional simulations done to determine the maximum possible profit (optimum card allocations) provide a good reference point for the performance of all the strategies. Five cases selected for simulation represent a wide range of production scenarios. In some cases one static control policy completely dominates the other, in others they outperform each other in particular R/C ranges. For instance, in case 1 one-card kanban is superior for low R/C while ten-card kanban performs better for high R/C. In cases 2 and 3, one-card kanban is without a doubt the winner whereas in case 4 ten-card kanban is clearly the better performer. Thus, a multitude of cases displaying a variety of performance patterns provide a solid foundation to test the adaptive control strategies developed.

The inspection of simulation results shows that the adaptive strategies *buffer_size* and *delay_time* consistently perform well. In cases 1, 2 and 5 both generate more profit than either static control policy. In extreme cases tailored specifically to the strengths of the static policies such as

cases 3 and 4 they stay within a few percent of the best performer. The bottom line is that they are always near the top performance-wise. The same cannot be said for static control policies however. For instance in case 3, a case which the conservative one-card kanban is best suited for, ten-card kanban fails catastrophically; it actually ends up losing money (negative profit). Similarly, in case 4, which favors aggressive control policies with high throughput such as ten-card kanban, the conservative one-card kanban falls far behind.

It is however the mixed case which solidifies the position of *buffer_size* and *delay_time* as efficient and consistent control policies. According to the simulation results both policies perform consistently good in a wide range of R/C values.

Between the two adaptive control strategies, *delay_time* appears to be doing better overall. As discussed before, this can be attributed to the smaller step size used by this policy. Pre-learning allows adaptive policies to start the runs with good control parameter values and enables them to stabilize the system rather quickly. Once operating near optimum, *delay_time* with smaller step size can better track the small variations in system parameters.

Another observation made is that *delay_time* in a majority of cases is more conservative than *buffer_size* in terms of throughput. It produces less parts but also pays less inventory cost. By the same token, both are generally more conservative than the best kanban policy.

4.6 Observations on Adaptive Behavior

The simulation results presented in previous sections show that the adaptive control strategies *buffer_size* and *delay_time* have the ability to adapt to the changing conditions in the manufacturing environment and perform successfully in a wide range of manufacturing setups. They can compete with static control policies even in setups that are specifically designed for those static policies to be successful. What makes the adaptive strategies *buffer_size* and *delay_time* to adapt and learn is their rule-based learning heuristics. This section is dedicated to the analysis of these heuristics to gain insight on adaptive behavior. Elements of adaptive systems are identified and several observations on adaptive behavior are shared with the reader.

As explained before, the stations comprising the production line complete a virtual manufacturing session before the real production starts. In this virtual manufacturing session, which serves as a training tool, the stations act as if they are in real production and send each other messages over the network. This virtual production phase is called pre-run. During pre-run stations operate in pre-learn mode, that is, they use randomly generated production parameters (service time etc.) and record all the necessary information pertaining to their operation. In this intelligent manufacturing environment each station knows and keeps a record of its own production parameters.

The purpose of pre-run is to come up with a good guess for the control parameters and initialize the system at a proper operating point. Since the system is already at a near optimum operating point at the start of the actual run the inspection of the behavior of the system during the actual run, will not give us many clues about the learning mechanism. On the other hand, if the manufacturing system is observed during the pre-run, where the system goes from zero to near optimum, the key aspects of learning heuristics will be identified more clearly. For that reason the pre-run in case 3 (severe bottleneck in Section 4.4.3) is analyzed below. This case is particularly suitable for analysis because the steady-state operating point is far from the zero operating point, that is, the state of the system at the start.

4.6.1 An Example

The next three figures belong to the pre-run of learning heuristic *delay_time* in case 3. Figure 4-18 is the plot of profit during the entire pre-run (five million time units, 500 intervals) whereas Figure 4-19 is a more detailed plot that covers only the first 100 intervals. Plots show that it takes the system about 60 intervals to reach the steady-state. Considering the fact that the duration of the actual run is 100 intervals the system requires a long time to adjust itself to the new operating conditions. Once the steady-state is reached, the overall profit remains nearly constant with the exception of a small fluctuation at interval 300.

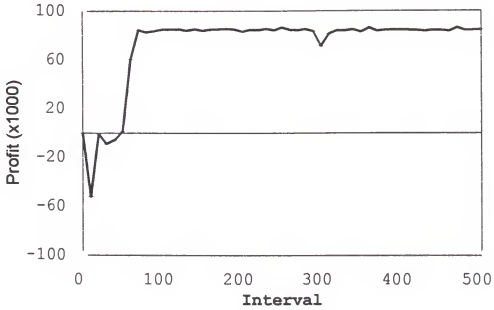


Figure 4-18. Profit in pre-run of heuristic *delay_time* in case 3 (R/C=1000).

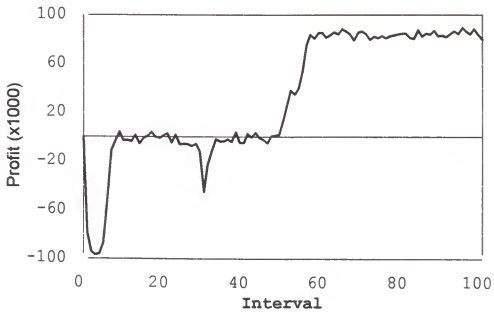


Figure 4-19. Detailed plot of profit in pre-run of heuristic *delay_time* in case 3 (R/C=1000).

The comparison of the plot of the control parameter (Figure 4-20) to the plot of profit reveals that the profit stabilizes as soon as the control parameters for S2 and S3 reach their steady-state value. Figure 4-20 shows that S2 adjusts quickly by going from zero to 20 in less than 20 intervals, however it takes S3 much longer to adapt. One reason is that it is the bottleneck station; it requires a longer wait period to be profitable. The other reason is the small step size (2 time units) of the control parameter. The input buffer of S3 fills up quickly causing it to operate with maximum inventory. Even when S2 adjusts by applying a long delay of its own to parts entering its buffer (at interval 20) thereby limiting the input to S3, S3 still can not handle the load and struggles to consume the parts

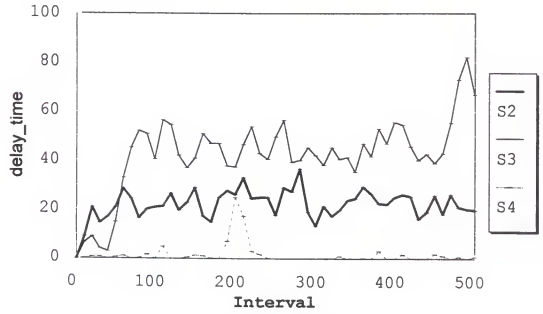


Figure 4-20. Control parameter in pre-run of heuristic *delay_time* in case 3 (R/C=1000).

already occupying its buffer. As a result, the learning heuristic comes to the conclusion that increasing the wait period was not such a good idea and does just the opposite. This causes the wait period to remain low and the system to lose profit. Even when *delay_time* of S3 finally starts going up (at interval 40) it takes 20 intervals for it to reach the desired value.

This analysis shows that the disadvantage of having a small step size is twofold. First, the randomness in the system or pre-existing conditions may fool the learning heuristic if the changes in control parameters (step size) are not large enough to make a difference. In other words, the heuristic does not get the good feedback that it needs in order to work properly. Secondly, even if the heuristic correctly predicts the direction, it may take a long while for the system to stabilize if it is far from the desired operating point.

4.6.2 Step Size

One of the issues in learning is the determination of the step size used by the learning heuristic. As explained before, the learning heuristic manipulates the control parameter after every interval, either increasing or decreasing it by an amount referred to as the step size. How learning is affected by step size is one of the key topics to be discussed.

In control heuristic *delay_time,* the material flow is controlled by varying the control parameter which is defined as the amount of time a station makes each part wait before it enters its input buffer. In the simulations presented, the

duration of the mandatory wait administered to the incoming parts is increased or decreased by 2, that is, the step size is 2. In control heuristic *buffer_size*, learning is facilitated by varying the buffer capacity of each station.

The analysis of the simulation results reveal that the differences in learning patterns of the control strategies *buffer_size* and *delay_time* result not only from the different control mechanisms they employ but also from the different step sizes they use.

The disadvantages of a small step size was discussed in Section 4.6.1. It was stated that the small step size generally makes it difficult for the system to recover if a sudden change in setup parameters causes a significant shift of the desired operating point. Even if the learning heuristic is able to correctly determine the direction to go, it will still take a long time to get there. An example to this is the mixed case with control strategy *delay_time* in Section 4.4.6. Figure 4-14 indicates a slow recovery period at the beginning of the second and third quarters caused primarily by the slow increase (decrease) of the control parameter as shown in Figure 4-15.

Moreover, the random nature of the operation parameters or residual effects of the pre-existing conditions might fool the learning heuristic. Learning requires proper feedback. To ensure proper feedback, the corrections made by the learning algorithm must generate a clear and detectable response. A small change in the control parameter however might not be sufficient to cause a

large enough change in the output. Especially if the system contains large amounts of noise due to either random fluctuations in parameter values or residual effects, the learning heuristic might not get the correct response from the system and might consequently make the wrong decision. This would at best delay the system from reaching the desired operating point. In the worst case, the system settles in a non-desirable operating point and operates inefficiently. An example to this is the case illustrated in Figures 4-19 and 4-20. Between intervals 10 and 50 the system gets stuck in a non-near-optimal operating point because the learning heuristic makes the wrong decision and the control parameter of S3 remains at a low value.

Employing a small step size has its advantages as well. Due to the step sizes involved, an increase or decrease in wait period naturally has a much smaller effect on the overall performance of the production line than an increase or decrease of buffer capacity. *Buffer_size* is at a disadvantage because the minimum possible step size it can use is 1, and even that, in certain situations, will be enough to change the performance drastically. On the other hand, *delay_time* can follow the tendencies of the other stations much better because it can modify its control parameter in small increments. Perhaps this point can best be illustrated with an example. The production line described in case 3 has no tolerance for high inventory levels. The bottleneck station S3 limits the material flow in

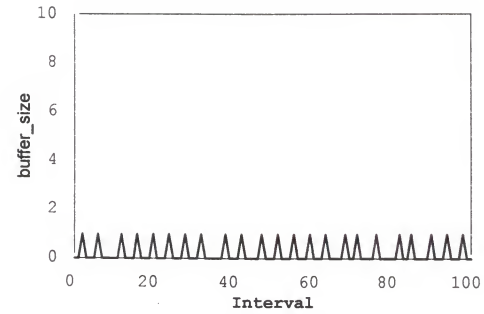


Figure 4-21. Variations in control parameter *buffer_size* of S3 in case 3 (R/C=500).

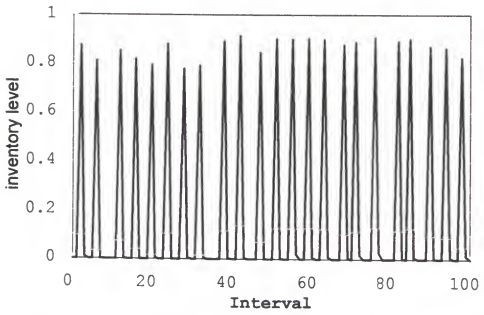


Figure 4-22. Variations in inventory level in S3's input buffer in *buffer_size* in case 3 (R/C=500).

the system making it infeasible for all the stations except its downstream counterpart S4 to operate with a buffer size larger than zero. In other words, stations should not accept parts unless they are starved. Thus, *buffer_size* policy is expected to imitate one-card kanban and yield similar results. However, the plot of profit percentages in Figure 4-6 shows that especially for low R/C, *buffer_size* is well behind the best performer one-card kanban and also behind *delay_time*. It can be speculated that this is caused by the high average buffer levels *buffer_size* possesses at low R/C even though it starts the run at zero buffer capacity. The reason for this discrepancy lies in the fact that the learning module has to continuously search for

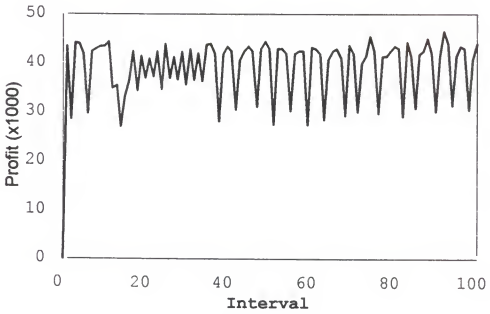


Figure 4-23. Changes in overall profit over time in *buffer_size* in case 3 (R/C=500).

a better alternative. As a result, the buffer capacity toggles between 0 and 1 as shown in Figure 4-21. The inventory build-up that occurs when it is 1 is responsible for high average buffer levels and the loss of profit. The up and down movement of the buffer capacity is what causes the jigsaw shape of the inventory plot in Figure 4-22. Every time the buffer capacity goes up to 1 a significant increase in the inventory level is observed in that interval. Note that in Figure 4-23, which shows the variations in the overall profit over time, the valleys correspond to the intervals where buffer capacity is 1 and the buffer levels are high.

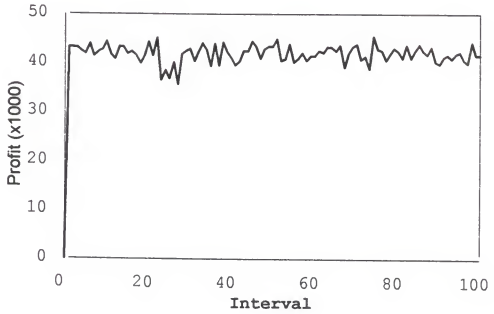


Figure 4-24. Changes in overall profit over time in *delay_time* in case 3 (R/C=500).

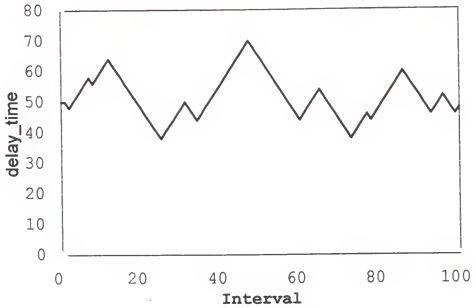


Figure 4-25. Variations in control parameter *delay_time* of S3 in case 3 (R/C=500).

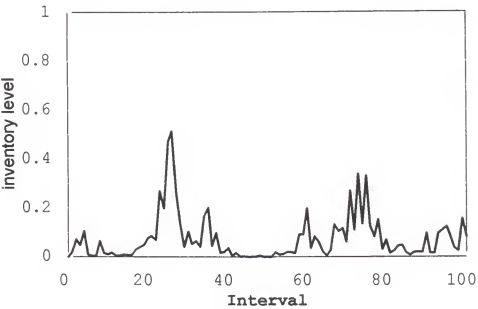


Figure 4-26. Variations in inventory level in S3's input buffer in *delay_time* in case 3 (R/C=500).

Figure 4-24 above is a plot that shows the variations in profit of a production line employing the *delay_time* heuristic. The comparison of Figure 4-23 to Figure 4-24 clearly shows that the dynamic range of fluctuations in *delay_time* are much smaller compared to *buffer_size*. The reason is that the control parameter *delay_time*, due to the small step size it uses, can adjust to the minor changes in operation conditions better. It should be noted that *delay_time* operates with a much smaller amount of inventory and the hills in Figure 4-26 which indicate high WIP correspond to low values of *delay_time* in Figure 4-25.

4.6.3 Oscillatory Behavior

One interesting observation is that in the plots above each control parameters appears to oscillate even in steady-state around an average value. The reason for this lies in the way the learning heuristic operates. The learning algorithm continuously searches for a better solution by modifying the control parameter throughout both the virtual and the actual runs. Even when the system reaches a state which yields a bigger profit than any other state before, the search does not stop. In a sense, the learning heuristic becomes never satisfied. This prevents the system to get settled in a less than desirable operating point. For instance, Figure 4-19 shows that in case 3 the profit is steady between intervals 10 and 50. Similarly, the control parameters for all stations also appear to oscillate around a constant value (Figure 4-20) between the same intervals. Without continuous searching, the learning algorithm would decide that the system reached the steady-state by looking

at this data. This would cause the production line to operate very inefficiently.

CHAPTER 5
DESIGN AND IMPLEMENTATION

A serial production line model is presented in Chapter 3. The distributed control aspects of the production line are studied. A rule-based learning approach is shown to offer promising performance comparable to the more traditional approaches to MOC. This chapter deals with the actual implementation of the production line model and its distributed control mechanism. The success of the design and the physical implementation depend a great deal on the selection of proper tools and technologies.

5.1 Technology Issues

As discussed in Chapter 2, the main elements of distributed control are the embedded processing capability and the ability to transfer data and messages over a network of nodes. The primary goal of this research is to demonstrate that it is feasible to apply distributed control to manufacturing operations. Since this is a study in the field of industrial engineering, issues such as cost-effectiveness, reliability and maintainability are of great concern. Thus, these design issues must be kept in mind when choosing an embedded controller and a network protocol to design

the distributed manufacturing system. We start with the selection of a communication protocol.

5.1.1 Communication Protocol

Concern for the real-time [88] aspects of a communication protocol is an important criterion. In embedded control, real-time refers to the case where the system being controlled does not wait for the data processor. In practice, real-time control is achieved by having data processing that is several orders of magnitude faster than the mechanical operations performed by the system. Motion control in machine tools is a good example. The calculation of the tool path and the tool velocity (vector) at each point of the path must be computed while the tool is in motion. However, the tool may not be paused to wait for computation results.

In distributed manufacturing operations control (DMOC), the communication protocol selected should be able to support high transmission speeds. DMOC messages fall into several categories. In a typical implementation, one may envision command messages (e.g., unload a part), information request and reply messages (e.g., is there room in the following buffer?), and bookkeeping messages (e.g., the number of parts produced in the workcycle). The availability of a prioritization mechanism that will allow the classification of high-priority messages, such as command messages, is highly desirable. Messages carrying time-critical information should have priority over other messages when gaining access to the communication medium.

Another important issue is reliability. To ensure the reliable operation of the network, the communication protocol must be equipped with a strong error detection mechanism. Cost-effectiveness and availability of parts are also of importance.

There are many communication protocols available, however the controller area network (CAN), which was developed by Bosch GmbH in the 1980's and quickly became an industry standard, appears to be the most suitable protocol for the task at hand.

CAN has many desirable features:

(a) Mature standard: CAN has been around for more than ten years and is supported by many chip manufacturers and vendors. Currently there are numerous low-cost CAN products and tools on the market.

(b) Hardware implementation of the protocol: The CAN protocol is implemented in silicon, often as a peripheral to a microcontroller. Such on-chip CAN controllers handle the transmission and reception of messages as well as error handling and fault confinement. This not only increases the message processing speed and performance, but also reduces the amount of code and facilitates application development.

(c) High speed and large bandwidth: In real-time applications where the timely delivery of messages is critical, high speed is an essential ingredient. Over short distances, CAN can reach transmission speeds up to 1 Mbit/s. In addition, the conflict resolution method (non-destructive bit-wise arbitration) employed by CAN greatly increases the

bandwidth in highly congested systems. It guarantees the continuity of transmission in the case of a collision. The term "collision" refers to the case where more than one message attempts to gain access to the network at the same time. In a collision, one of the colliding messages gains access to the bus as opposed to, say, Ethernet, where all messages have to get off the bus and re-try, which results in bandwidth loss. Moreover, in Ethernet and similar destructive collision protocols, bandwidth loss increases exponentially with network congestion. Bit-wise arbitration also allows the setup of a prioritization scheme that ensures fast transmission of high-priority messages.

(d) Simple transmission medium: CAN uses a multi-drop topology. Most commonly used transmission medium is twisted pair of wires. All nodes are connected to the pair of wires, similar to appliances plugged into wall sockets.

(e) Excellent error handling and fault confinement: There is an extensive error detection mechanism in CAN. CAN implements several mechanisms, both at the message level and at the bit level, to detect errors. CAN also has a unique feature called "error confinement" which prevents faulty nodes from disrupting the operation of the network. In a CAN network, nodes are self-aware and have the ability to distinguish between temporary and permanent failures. When a faulty node detects that it has a permanent failure it takes itself offline.

For reasons stated above CAN has been chosen as the communication protocol for this study. For a more detailed discussion on CAN the reader can refer to Appendix A.

5.1.2 Embedded Controllers

The other technology used in implementing DMOC is the embedded controller technology. The selection of the embedded controller depends to some extent on the communication protocol chosen. Today there is a number of microcontrollers in the market with a built-in CAN controller. The on-chip implementation of CAN and cost-effectiveness are important considerations in the selection of an embedded controller.

The issue under question in this study is cost-effective feasibility. Thus, we seek low-cost ($10-$100 per node) solutions. A suitable choice is the Siemens C167 microcontroller [65], also manufactured by ST Microelectronics as the ST10F167 microcontroller. It is a member of 16-bit CMOS single-chip microcontroller family designed specifically for embedded control applications. It provides a good combination of performance and cost. Siemens C167 is a low-cost ($5-$10) microcontroller with a CPU clock of 20MHz. It can execute up to 10 million instructions per second (10 MIPS) and features a sophisticated timer unit that provides up to four independent time bases. More importantly, C167 has a well organized on-chip CAN interface that provides full CAN functionality on up to 15 full sized message objects (eight data bytes each). That is, although the microcontroller's CAN unit appears as a single physical node on the network,

internally, it has 15 logical CAN nodes, each with its mask (address) and data registers. The availability of a large number of message objects enables the design of a versatile messaging scheme and provides flexibility for the application engineer. In addition, it provides masks for acceptance filtering which allows the objects to filter messages not relevant to them. That is to say, the message objects already posses a level of autonomous intelligence. They screen all the traffic on the network and interact with the ones specified in their masks. All this is done without the intervention of the microcontroller, and thus, with no application code.

The integrated CAN module handles the completely autonomous transmission and reception of CAN frames in accordance with the CAN specification V2.0 part B (active). In other words, C167 can receive and transmit standard frames with 11-bit identifiers as well as extended frames with 29-bit identifiers.

For more information on Siemens C167 microcontroller the reader is referred to the Siemens C167 manual.

5.2 Design Issues

As stated before, the distributed manufacturing system to be implemented is a serial production line. One of the decisions to be made is the size of the production line. Due to difficulties in modeling, a majority of analytical models in literature are limited to only two stations. A two-station production line is however not sufficiently rich for a study of DMOC. To be able to

analyze the interactions between the distributed components of the system, a production line consisting of a large number of stations is desirable. On the other hand, the complexity of the system can increase significantly if the size of the production line is not limited to a reasonable number. Consequently, this might lead to difficulties in the analysis of system behavior. Here it must be emphasized again that the objective of this study is to discover the underlying principles of DMOC. The fundamental principles can better be understood if the system analyzed is not overly complicated. Moreover, the design of a large production line will certainly create problems in implementation, such as wiring problems, increased setup time, complications in debugging, etc. Keeping in mind the main objective and considering the above mentioned practical design problems, a production line arrangement comprised of four stations is selected for implementation.

5.2.1 Physical Components of Implementation

In the implementation of the production line, each "smart" station is represented by an embedded controller. In the layout shown in Figure 5-1, all four embedded controllers are connected to the multi-drop CAN bus. All the communication in the network occurs through this CAN bus.

The arrangement depicted in Figure 5-1 is sufficient for the design of a real production line. However, the prototype built for emulation (Figure 5-2) requires an additional embedded

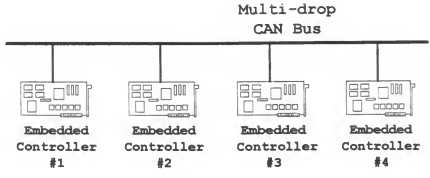


Figure 5-1. Layout of the emulated production line.

controller module, referred to as *scanner*. This additional unit has several purposes; it acts as an interface between the network and the host PC, it supervises the network, and facilitates the synchronization of the nodes. It also collects data to evaluate the system performance for the purposes of this study. In a real implementation, collecting research data may be left out.

Another component of the prototype shown in Figure 5-2 is the host PC. Programs run by embedded controllers are developed on this PC. A program is modified numerous times before it takes its final form and is ready to use. After each modification, the program is compiled on the host PC and then downloaded into the random access memory (RAM) of the controller. The modified program is tested and the results are noted. This procedure, often referred to as the "embedded code development cycle," is repeated continuously until the program is finished. The download

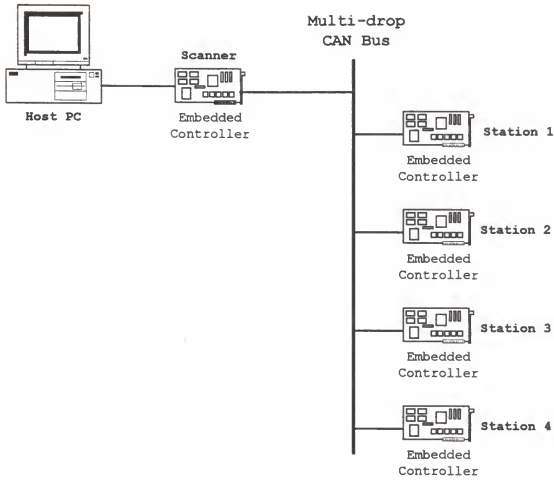


Figure 5-2. Test setup used for the emulation of the production line.

occurs via the serial port, that is, the serial port of the host PC must be connected to the serial port of the controller through a serial cable. The next step is to copy the fully functional program into the permanent read-only-memory (ROM) where it is likely to stay throughout the useful life of the controller.

5.2.2 Scanner Module

The first responsibility of the scanner module is the initialization of the network. Before the emulation starts, it

performs a series of handshakes and establishes contact with all the nodes in the network. Also among its tasks is the periodic collection of data from the nodes in the network.

The scanner records production statistics and passes them on to the host PC. To facilitate learning, a production run is carried out as a succession of shorter production cycles called "intervals". As explained in Chapter 3, during an interval each station collects data pertaining to its operation. Based on these data, it determines the necessary changes to its operation policy for the next interval. The collected data is also sent to the scanner after every interval. Upon the request of the scanner, each node transmits its operational data such as parts processed, profit, and so on. The scanner module records the data it receives and sends it to the host PC to be displayed on the screen.

In a sense, the scanner acts as a physical as well as a logical bridge between the stations and the host PC. In more sophisticated applications, the PC-scanner-network structure can be used to enable human input to the system. Using the PC, the engineer can interact with the system during production. The operator can instruct the network to override its mode of operation, by switching to a new control parameter, modifying the set of operation rules used, or altering the learning mechanism employed.

The synchronization of the components of the network is an important issue that needs to be addressed. Each node in the

network has its own internal clock. As explained later, an interrupt-driven time base is established in each embedded controller unit to implement this clock. During an emulation run, it is important that all the stations operate in a synchronized manner. In other words, at any given time, all internal clocks must have the same value. Hence, a mechanism has to be developed to ensure the proper synchronization of stations.

The method employed in this design assigns this responsibility to the scanner unit. At the beginning of each production interval, the scanner unit sends an electrical signal (not a CAN message) that signals the start of that interval to each controller in the network. Similarly, the scanner unit signals the end of a production interval by sending another electrical signal. This way, all stations start and stop their operation at the same time. Since a production run is divided into many intervals, this ensures frequent synchronization of individual clocks.

5.2.3 Time Base

Perhaps the most fundamental design step is the construction of a time base. Stations need to be aware of time to coordinate their operation. This is more so for the test setup (prototype) designed here than the actual manufacturing setup. The prototype production line conducts a virtual production based on artificially generated events. Processing of parts, breakdown and repair of stations are not real events, they happen only in the "mind" of the stations. A random number generator (RNG) is used

to determine the duration of these events. For example, as soon as a station breaks down, its status is changed to "down" and the random number generator program is activated. The number it produces determines the repair time. Then the station, that is, the embedded controller, starts counting the clock ticks. When the repair time winds down to zero, the embedded controller restores the status of the station back to "working" state. The diagram in Figure 5-3 illustrates how this status change occurs in a station.

A timing mechanism needs to be devised to simulate the passing of time and to generate the clock ticks necessary for

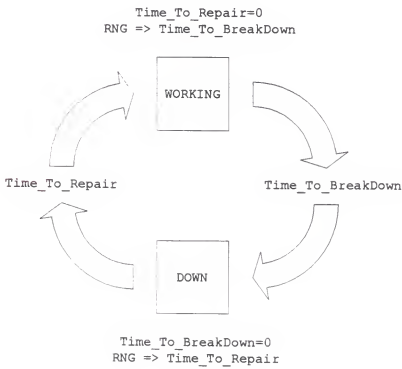


Figure 5-3. Status change at a station.

successful simulation of artificially generated events. This mechanism involves the design and implementation of an internal clock at each station. The embedded controller unit of each station uses its timer circuitry to periodically generate an electrical signal, which serves as a clock tick. At every clock tick, the program running in the controller updates the status of the station.

Naturally, the frequency of the clock ticks determines the speed of the emulation. If clock ticks are generated at a very fast pace, the virtual production is completed in a short amount of time. However, there is a finite limit as to how frequently they can be generated. A certain amount of time is necessary to run the program itself that controls the operation of the station. For successful operation, a clock cycle (duration between two clock ticks) may not be shorter than the amount of time required for the program to complete reviewing all its inputs and determining all its outputs. This process of reviewing the inputs and generating the corresponding outputs is often called a loop time in industrial control. It is also referred to as a cycle, a term that introduces confusion in the current context.

Transmission of CAN messages also takes time. Instantaneous message transfer implies that the CAN message arrives at its destination during the same clock cycle that it is transmitted. Consequently, to ensure instantaneous message transfer, enough time must be allowed between clock ticks.

5.2.4 CAN Messages

Communication in the network occurs via transmission of messages over the CAN bus. Each node in the network is capable of transmitting and receiving CAN messages.

The microcontroller's CAN unit employs a special mechanism called message object to coordinate and regulate reception and transmission. A message object defines the necessary parameters for successful transmission or reception of a particular message, such as arbitration ID, message direction (receive/transmit) and data length. Therefore, every message that a node receives or transmits must be associated with a message object.

Message objects serve as storage for incoming and outgoing CAN messages as well as for configuration and control data pertaining to them. A message object consists of a message control register, a message configuration register, an arbitration field, and a data field. Before a CAN message is transmitted, its data is placed in the data field of the message object it is linked to. The transmission is initiated and controlled via the message control register. Similarly, a received message is placed in a message object as well. The arbitration field contains the arbitration ID of the message received, the data field stores the data and the control register reports the current status.

The association of a CAN message and a message object is established via the arbitration ID. When a message is received,

the CAN controller compares the arbitration ID of the incoming message to the arbitration registers of active message objects in the node. If a match occurs, the message is stored in the matching message object. If there is no match, then the CAN controller, hence the node, ignores the message. In computer science, this type of data storage is referred to as "Content Addressable Memory (CAM)" since the physical storage location is extracted from the contents of the data (message). Before the production starts, message objects must be properly configured. Therefore, the configuration of message objects is an important task.

To facilitate message transmission, each station is given a station identification number (station ID). Some of the messages are broadcast, that is, all stations are eligible receivers, other messages are addressed to specific stations. If a CAN message transmitted has one destination, the station ID of the targeted station can be specified in the message.

The messaging scheme developed for the prototype design is intended to contain as much functionality as possible. It employs various types of messages to demonstrate all aspects of network communication. Some messages transfer data, others are commands and inquiries. On the other hand, to avoid unnecessary problems in the implementation, the number of messages and the amount of message transfer are restricted. This is done without sacrificing functionality. All communication in the network can be grouped into three categories: initialization (scanner-station),

production (station-station) and data transfer (scanner-station). Communication between the scanner and the stations occurs first during the initialization phase before the production starts, and then periodically between production intervals. During production however, communication in the network is limited to stations only.

5.2.4.1 Initialization messages

The purpose of initialization is to prepare the network for production. The initialization messages used in this design are *Request_ID*, *Transmit_ID*, and *Last_Node_ID*.

The initialization phase starts with the scanning of the entire network. This is done mainly to find out the number of stations connected to the CAN bus. In this particular design, the scanner needs a list of stations to be able to compile production data. Besides, stations need to know their position in the production line. Scanning also provides a mechanism to determine the operational nodes in the network.

The scanner unit broadcasts a wake-up signal to the network. As illustrated in Figure 5-4, it is a request sent to all stations in the network. The transmission of this CAN message (*Request_ID*), classified as remote request (see Appendix A), starts the initialization phase. Stations respond by transmitting their station IDs in a *Transmit_ID* message (Figure 5-5). Every time the scanner receives a *Transmit_ID* message it retrieves the

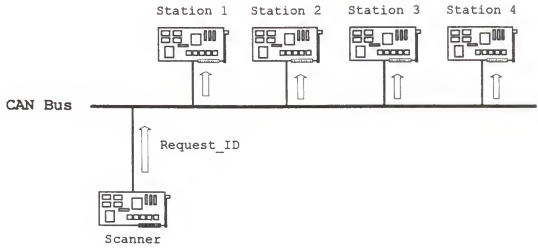


Figure 5-4. Message *Request_ID*.

station ID from the data field and adds it to its list of station ID's.

It is essential to the operation of the production line for the individual stations to know their position in the production line. This is so because the operation of the last station is different from the operation of an intermediate station, which is different from the operation of the first station. For instance, intermediate stations interact with both of their neighbors while the first and last stations have only one neighbor to interact with. Moreover, when the last station finishes a part, it sends that part to the finished goods storage whereas all other stations have to get permission from their downstream counterpart before they can unload that part.

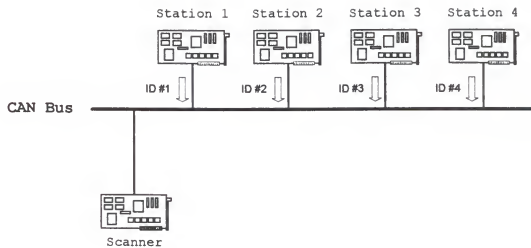


Figure 5-5. Message *Transmit_ID*.

Without loss of generality, it is assumed that the station IDs reflect the physical position of stations, that is, the first station of the production line is the station with ID#1, the second station is the station with ID#2, and so on. This way, the only information that stations need to determine their position is the ID of the last station. Thus, the last stage of the initialization phase is the transmission of message *Last_Node_ID*. The scanner unit completes the initialization of the network by broadcasting a message (Figure 5-6) that informs stations of the ID of the last station in the production line.

Further flexibility can be achieved by modifying the design to allow dynamic configuration of the production line. In such a design, the position of the stations can be re-arranged before or

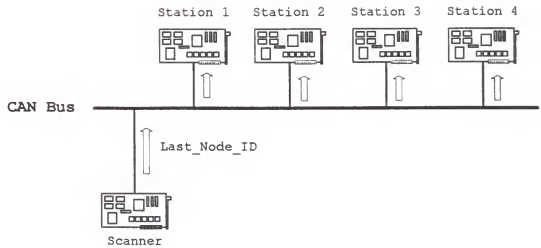


Figure 5-6. Message *Last_Node_ID*.

during production. This requires the stations to acquire the ID of the downstream station as well as the ID of the upstream station. Consequently, station IDs do not have to coincide with the physical position of stations anymore. An operator, using a PC, can download a new configuration to the scanner unit, which has the responsibility to contact every station and inform them of their new position. In this case, a new message that carries the downstream ID and the upstream ID must be added to the message structure.

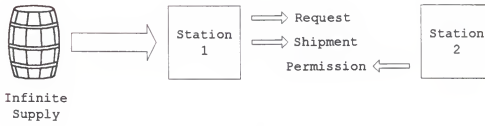
5.2.4.2 Production messages

At the initialization stage, stations prepare themselves to receive and transmit CAN messages. First, each station configures the message objects that it needs to communicate with the scanner, then it finds out its position in the production line (*Last_Node_ID* message) from the scanner. The next step is the

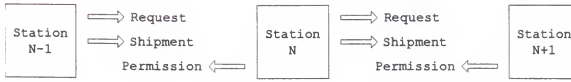
configuration of message objects necessary for interaction with other stations during production. These messages, called production messages, are *Request*, *Permission*, and *Shipment*. Figure 5-7 illustrates the interaction of stations during production. The first station communicates with only the second station (Figure 5-7a), therefore it uses only two transmit (*Request*, *Shipment*) objects and a receive (*Permission*) object. The intermediate stations on the other hand interact with two stations (Figure 5-7b). As a result, they use six message objects: three for transmitted messages and three for received messages. Finally, the last station exchanges messages with only the previous station (Figure 5-7c). Hence, it uses one transmit (*Permission*) and two receive (*Request, Shipment*) objects. Once the configuration of production message objects is completed, stations are ready for production.

Production starts with a signal from the scanner module. The details of part transfer between stations are given in Chapter 3. As explained there, a station cannot freely pass a processed part to the next station. A series of actions need to be taken by both stations for a successful part transfer. Figure 5-8 illustrates the protocol to be followed.

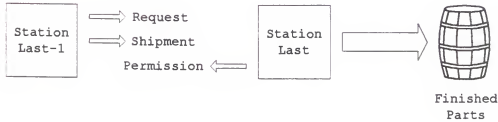
As soon as station N finishes the part it has been working on, it sends a message (*Request*) to station N+1 to notify that he is ready to unload. Station N also changes its status to "Blocked" (Figure 5-8a) immediately. It remains blocked until



(a)



(b)



(c)

Figure 5-7. Message traffic at various stations.
a) First station;
b) An intermediate station;
c) Last station.

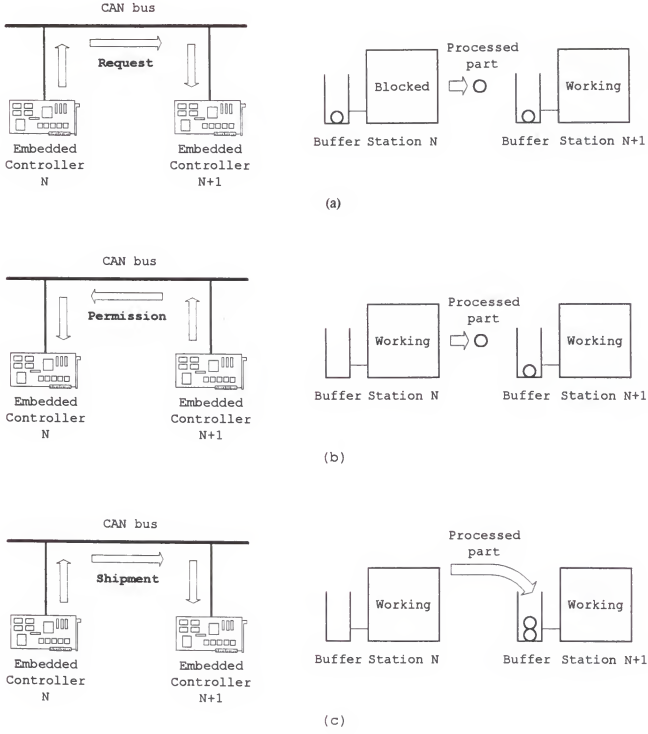


Figure 5-8. Protocol for part transfer between stations.
a) Station N requests to unload the processed part;
b) Station N+1 grants permission;
c) Station N unloads the part.

station N+1 decides that it is ready to accept a new part. As soon as station N receives permission from N+1 (Figure 5-8b), its status changes from "Blocked" to "Working." Consequently, it loads a new part and starts processing it. It should be noted that at this point, the processed part is still waiting at the output of station N. The part transfer occurs with the transmission of the message *Shipment* from station N to station N+1 (Figure 5-8c). In other words, the actual part transfer is simulated with the transmission of this message. As soon as the message (e.g. part shipment) arrives, station N+1 responds by increasing its buffer level.

Production messages contain no data in their data field. Instead, the arbitration ID field of a production message carries all information necessary, that is, the station ID of the transmitting station and the message type. Refer to Section 5.3.3 for a discussion on the classification of messages and on the selection of the arbitration IDs.

5.2.4.3 Data transfer messages

As described before, a production run is divided into many intervals. At the end of each interval, stations process the local data they collected, and based on these data they make decisions regarding their operating parameters. In other words, the period between two intervals is used for self-evaluation by each node.

In a real manufacturing system, the stations do not need to interrupt their operation to do self-evaluation. Both tasks,

production and self-evaluation, can be performed concurrently. However, in an emulation where the computing resources are already fully utilized, the additional load might be too much. Therefore, it might be a good idea to halt production after every interval, and concentrate on processing the information gathered. This brief stoppage can also be conveniently used to transmit some of the operational data collected to the scanner module.

The process of data transfer from stations to the scanner is quite straightforward. At the end of each interval, the scanner module sends a *Request_Stats* message to all the stations in the network, one by one. The station that receives the request immediately responds by sending a *Transmit_Stats* message that contains the desired information.

The data field of a CAN message can store up to eight bytes. In this design, the local data transmitted to the scanner are the production count, the inventory cost and the value of the control parameter (*buffer_size* or *delay_time*) in the past interval. The production count and the control parameter are sent as two-byte integers while the inventory cost requires four bytes. This uses all eight data bytes of data field. For additional data transfer, more then one message has to be sent.

5.3 Implementation Issues

Previous sections deal with various aspects of distributed production line design from selection of proper technologies to construction of a messaging scheme. This section focuses on the

actual implementation of the production line design. The following sections discuss various implementation issues that affect system performance. These include the selection of a data transfer rate and an internal clock speed which determine the emulation speed, the selection of CAN arbitration IDs and modes of communication which affect the complexity of the system, and the selection of a proper programming language which determines the development time and the size of the resulting code.

5.3.1 CAN Data Transfer Rate

In CAN, transmission of messages can be done at various speeds. The transmission speed is predicated on the length of the CAN bus. For example, to achieve the maximum data transfer rate of 1 Mbit/sec, the bus length must be limited to 40 meters. Even though the bus length used in the prototype is less than 40 meters, a more modest rate of 500 Kbit/sec is chosen to assure reliable message transfer.

5.3.2 Time Base

The time base, that is, the internal base clock, is implemented using the timer interrupt of the Siemens C167 microcontroller. The timer circuitry inside the microcontroller is programmed to generate an interrupt about every 10 milliseconds.

In case of an interrupt, the controller stops whatever it is doing (the current task is pre-emptively interrupted, and hence the term) and jumps to the interrupt service routine (ISR). An ISR is a special subroutine that contains the code to be executed

every time an interrupt occurs. In general, ISRs may be viewed as special subroutines asynchronously called upon a signal or an event, rather than called explicitly by a branching instruction. When activated, the ISR for the timer interrupt increments the clock counter that keeps the time. In a sense, the clock ticks every time the timer interrupt occurs, that is, every 10 milliseconds. This represents the time quanta, or smallest time quantity in the system. Consequently, time increments in steps of 10 milliseconds in the emulations. The reader is referred to Appendix C for the ISR code and the timer setup routine.

It must be pointed out that the emulation is a speeded up version of actual production. As mentioned above, one time unit corresponds to approximately 10 milliseconds. Hence, a production cycle that lasts one million time units takes only a little less than three hours. In real life however, this might correspond to six months' worth of production.

5.3.3 CAN Messages IDs

CAN version 2.0B is described in great detail in Appendix A. As explained there, each CAN message uses a 29-bit arbitration ID. Thus, a 4-byte (32 bits) register called the arbitration register is necessary to store the arbitration ID. Figure 5-9 shows the configuration of the arbitration field for all CAN messages in the production line design implemented in this study. The arbitration field consists of four bytes: byte 0, byte 1, byte 2 and byte 3. The symbol X in byte 3 indicates the unused bits in the arbitration register. As shown in Figure 5-9, a

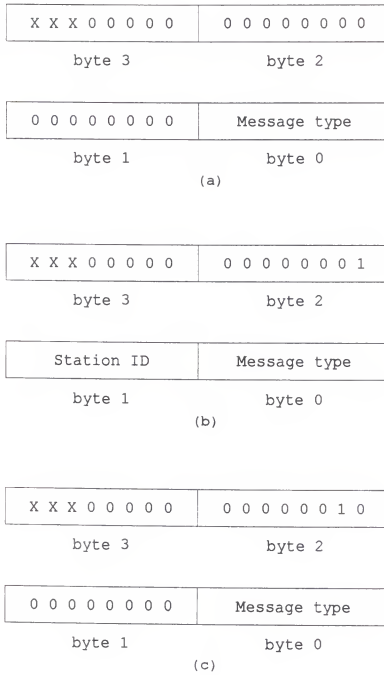


Figure 5-9. Identification of messages using arbitration IDs.
a) Initialization family messages;
b) Production family messages;
c) Data transfer family messages.

13-bit field made up from eight bits of byte 2 and five bits of byte 3 specifies the ID of the message family. According to the figure, family ID zero designates the initialization message family, family ID one designates the production message family, and family ID two designates the data transfer message family. It should be mentioned that this configuration allows 8192 (2^{13}) message families.

Table 5-1 provides a detailed listing of all CAN messages implemented in this design. It also illustrates the mechanism

Table 5-1. CAN messages and their arbitration IDs.

Message	Message Family	Arbitration ID (hex)		
		Byte 3-2	Byte 1	Byte 0
Request_ID (Remote Req.)	Initialization	00 00	00	01
Transmit_ID		00 00	00	01
Last_Node_ID		00 00	00	02
Request	Production	00 01	StationID	01
Shipment		00 01	StationID	02
Permission		00 01	StationID	03
Request_Stats	Data Transfer	00 02	00	01
Transmit_Stats		00 02	00	02

employed to determine the arbitration IDs. It should be noted that the ID values displayed are in hexadecimal.

The initialization message family has 3 members in the current implementation. However, this number can be increased to 65,536 (2^{16}) due to the combined 16-bit identifier space of byte 0 and byte 1.

In the production message family, the message structure is a bit more complicated. The message identifier specifies the transmitting station (source) as well as the message type. Hence, the 16-bit field made up from byte 0 and byte 1 is divided into two separate fields. Byte 1 is designated to the station ID of the source, whereas byte 0 is reserved for the message type. Since the 8-bit station ID field allows 256 (2^8) combinations, the maximum capacity of the network is 256 stations in this configuration. Similarly, the 8-bit message type field (byte 0) allows 256 different combinations too. Consequently, each station can send up to 256 production messages.

The data transfer message family consists of two messages but just like the initialization message family the maximum possible number of messages is 65,536.

5.3.4 Communication Modes

There are two methods for the microcontroller to be aware of newly received CAN messages. One is to periodically check for a new message and the other is to use the interrupt mechanism. In the first method, called "polling," the processor spends CPU cycles while waiting for the message to arrive and therefore

cannot perform another task until the message is received. The second method involves the use of interrupts. The message object associated with the incoming CAN message is setup such that the reception of the message causes an interrupt. When an interrupt occurs, the program branches to the corresponding ISR and processes the incoming message.

Polling is suitable if the node expects a message and need not perform any other task before it has the message. On the other hand, if the node operates in multi-task mode, that is, performs several tasks simultaneously, the interrupt method is the only viable solution.

As indicated in Table 5-2, messages that belong to the initialization family employ the polling method. Stations know that they will be contacted by the scanner, therefore they do nothing but wait until the scanner broadcasts the *Request_ID* message. The control program running in station modules continuously checks whether the *Request_ID* message has arrived.

The method (interrupt vs. polling) to be used at individual nodes is only one of the implementation decisions regarding the selection of a suitable communication mode. In addition to node level communication, the application engineer has to be also concerned with network level communication. An example is the selection of a proper data transfer protocol between the scanner and the stations. Questions such as "Who will initiate data transfer?," or "Will stations transmit their data concurrently,

or will they do it one at a time?" are legitimate questions that need to be answered.

In the initialization phase, stations respond to the *Request_ID* message by sending their station IDs in a *Transmit_ID* message. An interesting problem arises due to the fact that, initially, the scanner does not know the number of stations and their IDs. Consequently, it must prepare itself to receive station IDs from an unknown number of stations. In this implementation, a single arbitration ID is assigned to all the *Transmit_ID* messages. In other words, stations, when transmitting their station IDs, use identical arbitration IDs. Since *Transmit_ID* messages from all stations have the same arbitration ID, an attempt to simultaneously transmit by more than one station will cause an error in the CAN network. To avoid this, stations transmit their IDs in a pre-determined order. This is referred to as timed-delay response. In this design, the station with ID#1 has the highest priority. It transmits while all others wait. Stations take turns reporting their ID to the scanner until the scanner has a complete list of stations in the network.

An alternative method to coordinate data transfer from stations to the scanner module is polling (network-level). Polling is conceptually associated with a master-slave architecture. The node that requests data is called the master, and the nodes that are supposed to transmit data are called slaves. The master is in charge of the operation. The slaves may not send the data before the master asks for it. In this mode,

the master initiates a polling sequence. It contacts the first slave by sending a request. After the slave transmits the requested data, the master contacts the second slave. This goes on until all slaves are contacted and transmit their data. Between intervals, data transfer from the stations to the scanner happens in a polled fashion. The scanner, acting as the master, polls the stations (slaves) in the network one by one with a *Request_Stats* message, and the stations answer the call by sending their production data in a *Transmit_Stats* message. Both,

Table 5-2. Communication modes used by CAN messages.

	Source	Transmission Method	Target	Reception Method
Request_ID	scanner	broadcast	all stations	polling
Transmit_ID	station	peer-peer (timed-delay)	scanner	interrupt
Last_Node_ID	scanner	broadcast	all stations	polling
Request	station	peer-peer	station	interrupt
Shipment	station	peer-peer	station	interrupt
Permission	station	peer-peer	station	interrupt
Request_Stats	scanner	polling	station	polling
Transmit_Stats	station	peer-peer	scanner	polling

the master and the slaves wait until they receive the expected message.

The production family of messages is a good example for interrupt-driven communications. During production, stations cannot afford to wait for a permission (a *Permission* message) or a shipment (a *Shipment* message), the emulation program must continue to run. The program must continuously check and update the status of the station, send a CAN message to another station, and so on. Consequently, the reception of production messages is done in an interrupt-driven fashion. Reception of a production type message triggers an interrupt, the processor stops working on the current task and services the interrupt. In this implementation, servicing the interrupt means raising a flag, that is, acknowledging the incoming message by making a note of it. The reader is referred to Appendix C for details and the coding of the ISRs.

5.3.5 Coding

The success of the implementation depends heavily on the coding decisions made. Overall system performance, availability of system resources (e.g. memory), the scope of the application, and the user-friendliness of the development environment are factors that need to be taken into consideration when selecting the development platform and the programming language.

As stated numerous times in this text, the primary goal of this study is to find a low-cost alternative to current production control techniques. Consequently, cost-effectiveness

is a major design criterion. This leads to the use of a relatively low-cost embedded controller with limited processing power. Another desirable feature of the system to be implemented is small memory space. This limits the size of the code to be developed and dictates the use of an efficient compiler. Even though object oriented programming is the right approach for developing a simulation environment, it is not appropriate for embedded control [58] due to the size of code it generates. For this reason, in this project programming is done mainly in the assembly language of the Siemens C167 microcontroller and also in C.

5.4 Emulation Results

The networked embedded controllers were run and the result of these runs were compared to the simulation results given in Chapter 4. This comparison verifies the functional integrity of the physical implementation.

The production line is tested for the following five cases described in detail in Chapter 4:

(a) Case 1: No Bottleneck (Balanced Line),

(b) Case 2: Single Mild Bottleneck,

(c) Case 3: Single Severe Bottleneck,

(d) Case 4: Variable Bottleneck,

(e) Case 5: Two Bottlenecks.

For production parameters such as service time, repair time, reward, the values listed in Section 4.3 are used. In each case,

an emulation run of one million time units is performed. With a 10 millisecond time base, each run takes about three hours. Due to the long duration, emulations do not include a pre-run as opposed to the simulation examples described in Chapter 4. The buffer size for each station is set to one at the start of emulation. For comparison purposes, simulations are also carried out under the same conditions. Table 5-3 shows the difference in profit and production count between the simulation and emulation results. The results provide the empirical verification that the physical implementation actually performed as designed.

To analyze the effect of time base on emulation results, the production line is emulated using the setup of case 4 at five different clock rates: 15 milliseconds, 10 milliseconds, 5 milliseconds, 2 milliseconds and 1 millisecond. Figure 5-10 shows the stability of the emulated production line at different clock rates. Obviously, the system is most stable at 15 millisecond clock. As it is evident from the plot, as the time base falls below 10 millisecond, the profit starts to deviate from the initial value. These deviations can be attributed to delays in

Table 5-3. Comparison of the emulation and simulation results.

Error %	Case 1	Case 2	Case 3	Case 4	Case 5
Profit	1.1%	0.4%	4.9%	1.8%	6.7%
Throughput	2.3%	3.0%	3.1%	4.2%	0.3%

the reception of messages due to increased clock rate. That is, for shorter time base periods, more than one clock tick passes between the transmission and the reception of some messages. In other words, message transfer is no longer instantaneous at high speeds. At even higher clock rates, the stations cannot tolerate the increased message traffic and start to fail. At the clock rate of 2 millisecond, approximately one of every two emulations results in a system shutdown. Those that do not crash yield about 3% error in profit compared to the profit value at 15 milliseconds. Below 2 milliseconds, all attempts result in a system crash.

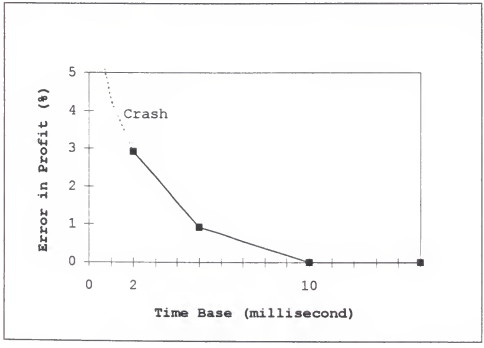


Figure 5-10. Change in profit as a function of time base.

5.5 Observations and General Remarks

All the functionality described in previous sections is achieved with a low-cost ($5-$10) 16-bit microcontroller with limited processing power (about 10 MIPS). The application platform is an off-shelf embedded controller unit without any operating system to help the application engineer with basic chores such as I/O and memory management. Moreover, only the most primitive debugging tools are available to detect and correct programming errors. In short, it is a minimal application environment without any luxuries that are available to a desktop application programmer.

The program running in station modules takes no more than 16 kilobytes. The program in the scanner module is even shorter, about 13 kilobytes. This is partly due to the fact that a large part of the coding, mainly the CAN setup routines, is done in the assembly language of Siemens C167 microcontroller. However, the size of the resulting program is still relatively small in today's standards. It is a quite interesting though not totally unexpected result of this work that a great deal of functionality can be derived from a 10-20 Kbyte program that runs in a low-cost microcontroller. Therefore, the main contribution of this study is that it demonstrates that the implementation of DMOC can be accomplished with limited resources. The results look more impressive when it is considered that the behavior of stations is determined by a simple set of rules and a straightforward learning mechanism.

5.6 Detailed Remarks on the Implementation of DMOC

A complementary goal of this research is to discover the empirical aspects of distributed control design. It should be pointed out that the success of the distributed design presented here and DMOC in general, hinges on the particular engineering approaches taken. No design, no matter how clever the underlying idea is, has a chance to succeed without engineering insight and sound engineering decisions. This research, being one of the first studies in this field of manufacturing, makes contributions by pointing out important design issues and drawing attention to possible implementation pitfalls. For this reason, emphasis is placed on specific problems and difficulties encountered during the design, implementation and testing of the prototype production line. These problems are documented, and solutions are suggested.

5.6.1 Issues in Simulating Distributed Systems

With the introduction of high performance PCs, simulation has become an affordable and valuable tool in engineering research and design. Simulations are generally preferred when the system to be analyzed is too difficult or too impractical to build. Simulation in this text is defined as the re-construction of a system in a virtual setting. More specifically, simulation involves a computer program that runs on a PC for the purpose of analyzing the behavior of the system it imitates. Its success, naturally, depends on how well it imitates the real system.

In industrial engineering, simulations are often used in the area of manufacturing systems design. These designs generally involve systems with passive elements (no data processing capability) and make no provision for information exchange between the elements. The concept of distributed manufacturing however, emphasizes local control and communication within the system. In a distributed manufacturing environment, such as the one modeled in Chapter 3, system performance is determined by the behavior of autonomous stations as well as the information exchange between these stations. Consequently, design efforts must concentrate on these issues.

For a meaningful analysis of any system, it is essential for the prototype design to reflect the fundamental characteristics of the real system. In a distributed system, this includes the presence of multiple data processing units and information exchange between them. For example, the development of a message structure that guarantees timely delivery of messages or the implementation of a time base are issues vital to the operation of the distributed manufacturing system. Unfortunately, these issues are usually left out in a simulation environment. Simulations almost always ignore transmission delays or loss of data, and consequently, do not investigate what kind of an effect these have on the operation of the entire system. Similarly, there are inherent limitations when it comes to the simulation of the concurrent operation of multiple nodes in a network, on a single PC. As a result, one arrives at the conclusion that

simulation, though an acceptable and sufficient design tool for systems with limited interactions, might be inadequate for DMOC. A realization of this study is that actually building a physical model significantly contributes to the behavioral analysis of a complex system such as the distributed production line studied in this research. The design and implementation issues have been discussed in great detail in previous sections. It must be pointed out that even this prototype does not truly represent the real production line. First of all, it relies on artificially generated events (breakdown, repair, etc.) rather than actual production. Secondly, it performs production at a much faster pace than the actual production line could. In short, it only emulates the real system. Despite these differences, the prototype provides a much more competent tool than simulation. It has all the functionality of the real production line; embedded controllers serve as elements of distributed data processing, CAN messages go back and forth through the CAN bus as they would in a real production line.

5.6.2 Observations from the Physical Implementation

As stated before, one of the goals of this study is to investigate the empirical aspects of DMOC. This part of the text is dedicated to the discussion of lessons learned from the physical implementation. Several observations are made and reasons behind important design decisions are shared with the reader. The reader should also keep in mind that some of the suggestions made and design principles laid out pertain to

specific technologies and components used (e.g., CAN, Siemens C167 microcontroller).

5.6.2.1 Communication rate

The first design issue to tackle is the selection of a proper data transfer rate for CAN messages. The maximum data rate allowed by the CAN protocol is 1 Mbit/s, however the actual rate depends on the length of the CAN bus. As the bus length increases, the transmission speed drops. While a data rate of 1 Mbit/s is achievable for a bus length of less than 40 m, the recommended rate drops to 500 Kbit/s for 100 m, and to 250 Kbit/s for 200 m.

In a factory setting, 1 Mbit/s is probably not achievable. Besides, the reliability of transmission is higher for lower data rates. Therefore, a 500 Kbit/s data transfer rate is selected for the emulations conducted in this study. It should be emphasized that all the nodes in the CAN network must be setup to transmit at the same data rate. Therefore, it is a good design principle not to change the CAN data rate once it is set. Manipulating it on the fly is very tricky and dangerous, and if not properly handled may cause some of the nodes to shut off or even the entire system to shut down.

5.6.2.2 Time base

A fundamental issue in the design of any emulation system is the construction of a time base, that is, the generation of clock signals. In particular, determining the duration of a clock cycle is a serious matter that deserves attention. A clock cycle is

defined as the amount of time between two clock ticks. Naturally, a fast internal clock, that is, a short clock cycle, is desired for fast operation. However, one must keep in mind that the emulation program that is running in embedded controller units requires time to execute its code. Naturally, the more sophisticated the code gets, the longer it takes to run. There should be ample time between two clock ticks to allow the program to check and update the status of the node it is running at. In other words, the program should be able to do all the housekeeping that it needs to do in that particular instant, before the next clock tick. Typical tasks include the generation of random numbers to determine the operation parameters, such as service and repair times, the update of node status, the reception of CAN messages, the processing of the messages received, and the transmission of CAN messages to other nodes in the network.

The sophistication level and the length of the code running in a node depends on the specific application. Therefore, it is difficult to make generalizations regarding to the amount of time that should be set aside for the execution of the program. On the other hand, certain assumptions can be made with respect to message transfer. The maximum length of a CAN version 2.0B message is around 140 bits. The actual length depends on the number of data bytes (0-8) the message carries. Assuming a fully loaded message and a data rate of 500 Kbit/s, the duration of a transmission can safely be estimated at one millisecond (ms).

This includes not only the propagation delay (~0.35 ms), but also the detection time and the process time. The next thing to consider is the amount of message traffic at a node. In the prototype design described in this work, an intermediate node interacts with both its downstream neighbor and its upstream neighbor. The part transfer protocol between two nodes involves the transmission and reception of three messages; *Request, Permission* and *Shipment*. Theoretically, an intermediate node may receive three messages and transmit three messages, all in the same time slot. Consequently, one time quanta, or one clock cycle must be long enough to allow the transfer and handling of all six messages. In addition, the transmission of some of these messages may be delayed due to additional message traffic caused by other nodes in the network. It is also always a possibility that one of the messages gets lost or corrupted during transmission. In that case, the corrupted message has to be re-transmitted. Factoring all of these into the calculation, a clock cycle of 10 milliseconds is selected in the implementation. It must be emphasized that expanding the size of the network by adding new nodes or diversifying the messaging mechanism by introducing additional messages can significantly increase message traffic in the network and lead to a much slower clock. Thus, an important lesson learned from the implementation is, that in general CAN message traffic rather than the housekeeping tasks performed at the node is the limiting factor as far as the speed of the emulation is concerned.

5.6.2.3 Synchronization

The presence of multiple internal clocks (one at each node) leads to a problem: synchronization of these clocks. For the proper operation of the entire production line, all stations must be on the same page in terms of timing. In other words, at any time, all clocks in the network must show the same value. To assure this, in the implementation described in this chapter all clocks are periodically synchronized by a master clock. The master unit, scanner, sends an electrical signal to start the production cycle at the beginning of each interval. All stations adjust their own clocks according to this signal. An electrical signal is preferred to a CAN message here, because it is faster and it is easier in terms of code development because it needs no additional code. Besides, it demonstrates an alternative communication method. However, in a real manufacturing system it would be better to restrict all communication to CAN. After all, the idea behind networking the manufacturing elements and the development of an industrial communication protocol such as CAN is to enable communication through a single, reliable and robust medium. The synchronization signal in that case would be a CAN message notifying all stations of the start or end of a new interval.

Another alternative for synchronization takes an entirely differently approach; instead of implementing internal clocks in individual units, one master clock can be used for the entire network. In this approach, the scanner module, which houses the

master clock, is responsible for generating the clock signals, that is, keeping time. Every time the master clock ticks, the scanner module sends a CAN message called "time stamp" to all stations in the network informing them that one time unit passed. For synchronous operation of the network, it is critical that all stations process the time stamp at exactly the same time. In many cases, even a small delay cannot be tolerated. This can perhaps be best illustrated with an example. Figure 5-11 gives a snapshot of the operation of a three-station production line. At time five, station A and B are processing while station C is starved. Stations A and B finish processing at the beginning of the next time unit, that is, at time six. Station B unloads the finished

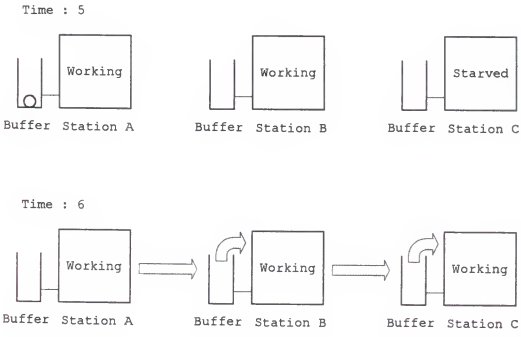


Figure 5-11. Synchronous operation in a three-station production line. All stations process the time signal at the same time.

part into station C's buffer which station C immediately starts to process. Similarly, station A transfers its finished part into station B's buffer which is immediately picked up by station B and processed. Therefore, at time six all stations are at state "Working." Now, let us consider the scenario depicted in Figure 5-12, where station B processes the time stamp later than the other stations. Upon the reception of the time signal, station A realizes that it is done processing and is ready to unload. However, at this point station B has still not processed the time signal. As far B is concerned the time clock is still at five and the part still needs more work. Therefore it does not permit station A to unload which causes A to get blocked. At the

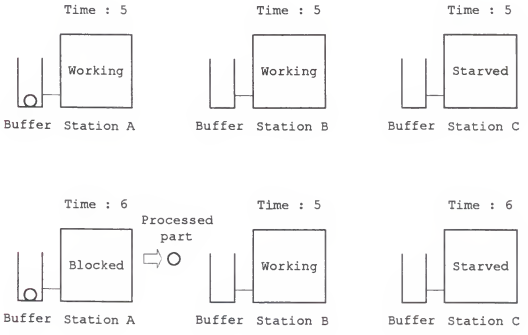


Figure 5-12. Disruption in the operation of a three-station production line due to the late processing of time signal by station B.

same time, station C remains in a starved state since station B is still busy processing. Consequently, the stations are blocked, working and starved, respectively.

The delayed processing of a time stamp can happen if the program running in the station is performing a high priority task at the time of the reception. If the program does not interrupt the current task and switch to the processing of the incoming time stamp, it will not be aware of the change in time. A solution to this problem is to assign the highest priority to CAN interrupts. In addition, the time stamp must be assigned the highest priority among all the CAN messages. This would ensure the immediate processing of the time signal received.

Another disadvantage of using centralized timekeeping is the increased message traffic in the network. The periodic transmission of clock signals might contribute to congestion in the CAN network.

It should be pointed out that the synchronization issue is more critical for an emulation setup than a real manufacturing system. In a real manufacturing environment, the difference between the transmission time and process/repair times is at least of several orders of magnitude. In a sense, time passes more slowly. It is not unreasonable to assume a time quanta of one second. As explained before, one can safely assume that it takes no more than one millisecond for a CAN message to travel from its source to its destination. Therefore, one second is a very long time for the transmission of messages. For all

practical purposes, all message transfer occurs instantaneously in a real manufacturing system.

5.6.2.4 Embedded code development

Programming in an embedded control environment requires an entirely different approach than programming in a desktop environment. A desktop programmer is used to having an operating system do all the housekeeping and not worry about things such as basic input/output, memory management or file manipulation. In addition, almost all desktop development tools come with sophisticated debugging tools. Unfortunately, a programmer working on an embedded control application does not have the same luxury. For instance, the development platform does not include an operating system or a real-time debugger. As a result, fundamental application development steps, such as debugging and testing becomes very difficult, cumbersome and time consuming. To be more specific, let us consider saving run-time data in a file, which is a well-known debugging method to programmers. Opening a file with a simple command is not an option in the embedded control environment used in this study. To store data pertaining to the operation of an embedded controller unit, one must format the data, specify a memory location and copy the data at that location. If a long stream of data is to be saved, these steps must be repeated as many times as necessary. It is also the programmer's responsibility to make sure that the memory location the data is copied at is available, that is, not occupied by some other data necessary for the execution of the program. To access

the saved run-time data after a test run the opposite is done; the memory location of the first data byte is specified, the data is copied from memory and sent via serial port to the host PC to be displayed on the screen.

Another method to study run-time data for debugging purposes is to print it on the screen during the test run. In this method, the embedded controller module (a station or the scanner) to be debugged is connected to the host PC via the serial port. The program running at that embedded controller is modified such that it forwards data describing the module's current state to the PC while virtual production is being conducted. The forwarded information is printed on the screen so that the programmer can follow the operation of that particular module. One drawback of this method is that the programmer can debug only one element of the production line at a time. There is only one serial link to the PC, therefore, only one module can communicate with the PC. In other words, it is not possible to observe the operation of more than one module at once.

Another potential problem associated with printing debugging data on the screen is that it might affect the operation of the node (station or scanner) being observed. If the programmer is not careful, the act of observation might change the way that node functions, which in turn, might disrupt the operation of the entire production line. There is a simple reason behind this; sending data serially takes considerable time. At a data transfer rate 9600 Baud, the transmission of one character from the

embedded controller to the PC takes approximately one millisecond. If polled serial communication routines are used, during that time, the microcontroller can do nothing else but oversee the serial transmission. Consequently, printing a data stream of 10 characters keeps the microcontroller busy for 10 milliseconds. Not to disrupt the operation of the system, the system clock must be slowed down enough to accommodate the transfer of data to the PC. Let us suppose that the system clock is setup to generate a clock tick every 10 milliseconds and a data stream of 10 characters is necessary to convey information about the status of a node every clock cycle. It must then be recognized that in this case 10 millisecond will be used just to send 10 characters to the PC. Consequently, the system clock must be slowed down to generate clock ticks every 20 milliseconds instead of 10 milliseconds. The bottom line is that it is not always possible to observe the system at high speeds.

As the discussion above shows, debugging a real-time embedded control application is not an exact science.

CHAPTER 6
SUMMARY AND CONCLUSIONS

This dissertation introduces the concept of distributed control in manufacturing operations and investigates its feasibility. From an engineering perspective, "feasibility" implies good performance, cost-effectiveness and reliability. The availability of necessary tools, technologies and expertise required for development are factors that must be considered and investigated as well.

The primary goal of this study is to demonstrate that distributed manufacturing operations control (DMOC) is not just an interesting idea but a strong candidate architecture in the future of industrial engineering practice. Of all the considerations of DMOC, perhaps the most important is its timeliness. The component technologies for DMOC have now become abundantly available. Similarly, the need for more "intelligent" MOC is becoming more pronounced, as industry seeks to strike some balance between the smarter manufacturing tools it uses and the ways it controls their collective operation.

Scientifically, this dissertation evaluates the hypothesis "It is feasible to achieve DMOC, comparable to the past traditional MOC techniques, using the newly emerged technologies." For this purpose, a specific manufacturing system,

namely a production line is studied. A small-scale manufacturing system emulated by a network of low-cost embedded controllers has been designed and physically implemented. In the process, technologies and tools necessary for this task have been determined. As a precursor to the actual low-cost implementation, numerous simulations were conducted to focus on to a specific DMOC architecture. This study resulted in a candidate approach, a simple rule-based learning technique. The physical implementation is an engineering effort to tangibly verify the capabilities of the proposed approach. Clearly, at this stage of academic understanding, questions such as the extensiveness of code, or the robustness of the proposed approach may be obtained only by experimentation. Such implementation is viewed as an important aspect of the effort, since this study employs the scientific method cycle of hypothesis and experimental evaluation, which in turn leads to a better understanding and further hypotheses.

The implementation verified the validity of the hypothesis for the chosen manufacturing systems. The prototype distributed production line proves to be a low-cost, reliable and compact design with a high degree of functionality. Moreover, the simulation results reveal that the DMOC design presented in this study, even with its modest operation rules and a straightforward learning mechanism, compares favorably to the more traditional MOC techniques.

Although, the hypothesis was an outgrowth of a strong engineering intuition that low-cost DMOC may achieve good

performance, the level of DMOC system performance emerging from rather simple nodes was undoubtedly the most unexpected result. For example, an embedded controller with 1MB of total memory was chosen for the task. The final code size was in the order of 10KB, two orders of magnitude below the expectation. This result is not only unexpected, but also very encouraging, since it is possible to implement significantly more sophisticated nodes with very modest code.

The results of this study contribute significantly to the current understanding of the capabilities of DMOC. They support the hypothesis, but more importantly, they suggest and encourage to many new directions of research. How will DMOC perform in manufacturing systems other than production lines? What will the impact of more sophisticated learning algorithms have on the performance of DMOC? What are the quantitative tradeoffs between node capabilities (say measured in MIPS) and communications capabilities (say measured in Mbit/sec). How may further layers of node intelligence be implemented? In this respect, perhaps the first investigation should examine how nodes may seek and determine their own operating parameters using not only local data, but also global data.

APPENDIX A
CONTROLLER AREA NETWORK (CAN)

The controller area network (CAN) communications protocol conforms to the seven-layer Open Systems Interconnection (OSI) model. The CAN architecture defines the Physical and Data Link layers of the OSI model. These two layers are transparent to the system designer and are included in any component that implements CAN protocols.

Physical Layer

CAN is a multi-master bus topology with data transmission rates depending on the overall length of the bus. For all ISO 11898 compliant devices the guaranteed speed is

- 1 Mbit/sec for bus lengths of up to 40 meters,
- 500 Kbit/sec at 100 meters,
- 250 Kbit/sec at 200 meters,
- 125 Kbit/sec at 500 meters.

Even though the number of nodes that can be connected to a CAN is theoretically unlimited, the drive capabilities of the available devices limit the maximum number to 32 or 64 depending on the device type.

A controller area network can be set up using various physical media such as twisted pair, fiber optics, and so on. even though the shielded twisted pair containing both the signal and power conductors is the most common. Flat pair (telephone) cable also performs well but may cause transmission errors at high speeds. For signaling, CAN uses differential voltages (voltage difference between the signal lines named CAN_H and CAN_L) which is the reason for its high noise immunity and fault tolerance. A *dominant* bit is defined as CAN_H being above CAN_L while a *recessive* bit represents the opposite case.

The Non-Return-to-Zero (NRZ) baseband encoding technique is the bit encoding method employed by CAN. It produces a minimum number of transitions (0 → 1 and 1 → 0) and therefore enables a higher transmission rate than the other encoding methods.

Its use of differential voltages together with its extensive error checking and its capability to operate in harsh environments allows CAN to continue to function even if one of the signaling lines is severed, or shorted to power or shorted to ground. The input comparators included in some CAN interface devices allow communication even if both lines are shorted together.

Data Link Layer

In CAN the data link layer is subdivided into

- a logical link control (LLC) layer which deals with the communication to the higher layers.

- a medium access control (MAC) layer which deals with message encoding and decoding as well as message prioritization, error detection and handles access to the physical layer.

CAN operates in multi-cast messaging mode. Data messages transmitted in a CAN do not contain neither the source address (the address of the transmitting node) nor the destination address (the address of the receiving node). Hence, more than one node can receive a transmitted message. Each message has a unique identifier field, which specifies the type of service this particular message provides (or the type of service it requests). By examining this identifier field, each node determines if the message is relevant to itself. Relevant messages are retrieved and processed, others are simply ignored.

Bus Arbitration

The real-time data transmission not only requires a high data transmission rate, but also an efficient bus arbitration mechanism to avoid excessive delays due to collisions. Additional to that, a message prioritization scheme is necessary to ensure the quick transmission of high priority messages (messages involving the breaking system should in general have a priority over those related to the cabin temperature control). For that purpose, the controller area network uses the well known *carrier sense, multiple access with collision detection (CSMA/CD)* with *non-destructive bit-wise arbitration* to provide collusion resolution.

As mentioned earlier, each message in a controller area network has a unique predetermined identifier, which is assigned during the design process. It is the value of that identifier that determines the priority of the message. The lower the identifier value gets, the higher its priority becomes. In general, a message with more *dominant* bits (0's) in its identifier has a higher priority. In case two nodes attempt to transmit on the bus simultaneously, the one with the lower priority message withdraws and tries again in the next bus cycle. Non-destructive bit-wise arbitration describes the procedure the nodes use to sense the presence of other transmitting nodes and to determine to stop or to continue their transmission. Both nodes, initially unaware of the other node trying to seize the bus, transmit the identifier bits one by one (starting from the most significant bit) to the CAN bus. Each node monitors the status of the bus while transmitting. The transmission by both stations continue until one node sends a dominant bit and the other sends a recessive bit. In that case, the bus transmits the dominant bit. The node that sent the recessive bit reads back a dominant bit and realizes that somebody else with a higher priority message is trying to transmit as well and quits. The other station does not notice anything and continues to transmit its message. Hence, the non-destructive bit-wise arbitration mechanism coupled with the uniqueness of identifiers guarantees the transmission of the highest priority data without delay.

The example below illustrates how the non-destructive bit-wise arbitration works. Suppose three nodes with identifier values 1436 (ID1), 1335 (ID2) and 1337 (ID3) are trying to transmit at the same time. The first three bits (bits 10, 9 and 8 below) of the identifiers are identical, therefore, all the nodes transmit these bits. Bit #7 for node 1 is recessive whereas the same bit is dominant for nodes 2 and 3. As a result, the dominant bit transmitted by nodes 2 and 3 overwrites the recessive bit of node 1. Node 1, realizing that there is another node with a higher message priority transmitting on the bus, respectfully resigns. Node 2 and 3 continue to transmit without noticing anything. At bit #3 however, node 2 transmits a zero (dominant), while node 3 transmits a one (recessive) as a result of which node 3 resigns and node 2 controls the bus alone.

As soon as a node loses the arbitration it becomes a receiver of the message being transmitted (as indicated by the dotted lines in Figure A-1) and suspends its own transmission until the bus becomes idle again.

Non-destructive bit-wise arbitration allows prioritization of messages and provides collision detection without minimum delays both of which are essential in real-time applications. Compared to token ring and classical CSMA/CD systems it proves to be superior. In a token ring architecture, which is based on the principle of time division multiplexing, a certain time slot is allocated to each node for transmission, which starts when that node receives a token. If the node has nothing to send it passes

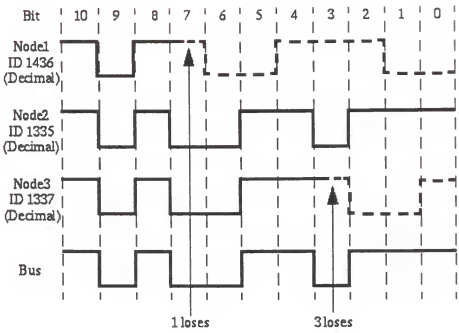


Figure A-1. Non-destructive Bit-wise Arbitration.

the token to the next node. In other words, every node uses a fraction of the bus time and controls the bus whether that it needs it or not. A node with an urgent message to send has to wait until the token makes its way back to that node which reduces the responsiveness of the system. On the other hand, the bus access is stochastic in Ethernet systems, which use CSMA/CD with exponential back off. In the case of collisions all nodes back off, wait for a random time and try to transmit again. In congested systems where a number of nodes try to access the bus simultaneously, the bus might remain inactive for long periods of time while all nodes fight to control the bus. This again leads to an inefficient network with excessive delay times.

Bit Stuffing

To generate a higher transmission speed the CAN protocol uses Non-Return to Zero (NRZ) bit encoding scheme which minimizes the number of signal transitions (0 → 1 and 1 → 0). However, transmission of messages consisting of long sequences of bits with the same polarity (all zeroes or all ones) might lead to the loss of synchronization. This is due to the fact that synchronization, which is explained later in this paper, is done in the falling and/or rising edges of a signal. Hence, synchronization is possible only if signal transitions occur in a message. To guarantee the necessary edges in the NRZ bit stream, thus to maintain the synchronization, a method called *bit stuffing* is used in CAN.

After the transmission of five consecutive bits of the same kind the transmitting station automatically inserts (stuffs) a bit of the opposite logic level. Therefore, the occurrence of more than five consecutive ones or zeroes in the transmitted bit stream is not possible. To prevent the corruption of the data, the receiving stations delete (de-stuff) inserted bits (the sixth bit) before processing the message. Bit stuffing is applied to all the fields from the start of frame (SOF) to the CRC delimiter.

CAN Frame Format

The CAN protocol employs the following types of frames:

- data frame
- remote frame
- error frame
- overload frame
- intermission frame

Data frames carry data (sensory input or control commands) from the transmitter to the receiver(s). Remote frames are transmitted to request a data frame (for example, to initiate the transmission of some measurement data). An error frame can be sent by any node that detects a transmission error, to inform the network of that error. An overload frame is sent by a node to indicate that it is not ready to receive data. Interframe space provides a space between successive data or remote frames.

CAN data link protocol defines two data frame formats:

- standard CAN (Version 2.0A)
- extended CAN (Version 2.0B)

The two formats differ mainly in that the extended format uses 29 bits compared to the standard format's 11 bits for the identifier field. The acceptance and widespread popularity of CAN has led to the idea of standardizing the assignment of message identifiers to certain communication functions. In that respect, a larger identifier field would give the system designer more options and freedom to define a well structured naming

convention. Hence, a new version is created which provides a larger address range with 29 bits available in the identifier field.

Old CAN controllers (version 2.0A) recognize only standard format messages. In the event of the transmission of an extended data frame they will flag an error. Some CAN controllers, known as 2.0B passive devices, do not generate an error upon the reception of a data frame of the extended format, but they ignore it. The new controllers (version 2.0B) implement both formats, i.e. they can send and receive both the standard and extended data frames without a problem. Because of this backward compatibility it is possible to use both version 2.0A passive and 2.0B controllers in the same network.

Standard CAN (2.0A) data frame format

A standard data frame consists of the following fields:

- start of frame (SOF)
- arbitration
- control
- data
- CRC
- acknowledgment (ACK)
- end of frame (EOF)

The start of frame (SOF) field consists of a single dominant bit, which marks the beginning of a data frame. The detection of

a dominant bit while the bus is idle indicates the start of a data (or remote) frame.

The arbitration field consists of an 11 bit identifier and an RTR (remote transmission request) bit. The identifier specifies the type of the message and its priority. The seven most significant digits of the identifier can not be all recessive. The reason for that is to provide compatibility amongst controllers from different manufacturers. The identifier 1111111XXXX is reserved to be used by the manufacturers to conduct their internal tests. The symbol X in the reserved identifier above represents any logical level (dominant or recessive). Therefore, the number of valid identifiers available for the use of the system designer is $2^{11} - 2^4 = 2032$.

A dominant RTR bit in the arbitration field indicates that the frame is a data frame while a recessive RTR bit implies a remote transmission request (remote frame).

The control field consists of six bits which includes a four bit long data length code (DLC) and two reserved bits (r1 and r0) for future expansion. The reserved bits r1 and r0 have to be dominant. The DLC specifies the length of the data field in bytes. The allowed length of the data field, which contains the actual data sent in the frame, is between 0-8 bytes. Therefore, the value stored in the DLC field can not be larger than eight.

The data field contains the actual data transmitted in a data frame. Its length is between 0-8 bytes as specified in the DLC of the control field.

CAN uses CRC (cyclic redundancy code) based frame check sequence for error detection. The frame check sequence in CAN protects the start of frame, arbitration, control and data (if present) fields of the frame. The polynomial used to derive the frame check sequence (called the generator polynomial) is

$$G(x) = x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$$
$$= (x+1)(x^{14} + x^9 + x^8 + x^6 + x^5 + x^4 + x^2 + x + 1)$$

The CRC field of the frame consists of a CRC sequence and a CRC delimiter. The CRC sequence is the remainder of the division of the bit stream made up from the protected fields by the generator polynomial. Since the generator polynomial is of degree 15 the length of the CRC sequence is 15 bits. To detect errors, the transmitted CRC sequence is divided by the generator polynomial at the receiver end; a nonzero remainder indicates transmission errors. The CRC delimiter is a single recessive bit, which marks the end of the CRC field.

The acknowledgment (ACK) field is two bits long and contains the ACK slot and the ACK delimiter. The slot bit is transmitted as a recessive bit by the transmitter and immediately overwritten by a dominant bit by the receiving node(s) if the receiving node(s) do not detect any transmission errors. The ACK delimiter is the second bit of the ACK field, which is sent as a recessive bit.

Finally, the end of frame (EOF) field demarcates the end of the frame. It is a flag consisting of seven recessive bits.

What follows the end of a frame is the intermission (INT) field consisting of three recessive bits. The intermission period gives the CAN controllers the necessary time to get prepared for another reception or transmission at the next bus cycle. After the intermission period, the bus enters the idle state if no other node starts a transmission.

CAN (2.0B) data frame format

The extended data format differs from the standard data format in the arbitration field. To the 11 identifier bits of the standard format an additional 18 bits are added to yield a total of 29 bit identifier field for the extended format. The 29 identifier bits provide the system designer with more than 500 million unique identifiers.

The first identifier field called the Base ID provides compatibility with version 2.0A. The two identifier fields are separated by an SRR (Substitute Remote Request) bit and an IDE (Identifier Extension) bit. The SRR bit is always transmitted as a recessive bit. Its purpose is to ensure that a standard data frame always has a higher priority than an extended data frame with the same base identifier (the first eleven bits of the identifier). The corresponding bit in the standard data frame is the RTR bit, which is transmitted as a dominant bit.

As explained before, a version 2.0B controller can transmit messages in both formats. When a version 2.0B controller transmits a data frame the IDE bit determines which format the frame is sent in. A dominant IDE bit indicates that the

controller sends a standard frame whereas a recessive RTR bit indicates the transmission of an extended frame.

The last bit in the arbitration field of an extended data frame is the RTR bit, which as in version 2.0A distinguishes between the data frame and the remote frame. A dominant RTR bit means that the transmitted frame is a data frame, whereas a recessive RTR bit implies the transmission of a remote frame.

Remote Frame

A node that expects a certain data can initiate the transmission of that data by sending a remote frame to the node that is expected to transmit that data. A remote frame is very similar to a data frame. The main difference is that a remote frame does not contain a data field. The type of the expected data frame is specified by the identifier field of the remote frame. In other words, the remote frame asks for the transmission of the data frame with the same identifier value.

A remote frame is identified with a recessive RTR (remote transmission request) bit. Also, the DLC (data length code) field of a remote frame is irrelevant since it has no data field. Aside from these the format of the remote frame is identical to that of the data frame (for standard and extended formats).

Error Frame

An error frame consists of two different fields. The first is the error flag whereas the second one is the error delimiter. Error frames are sent to notify the network of transmission errors. As soon as a node detects an error it immediately

transmits an error frame. It is important to note that the transmission of the error frame occurs during the transmission of the original corrupted message. The error flag field of an error frame consists of six consecutive bits of the same polarity. If the error flag is made up from dominant bits it is called an active error flag, otherwise it is called a passive error flag. An active error flag violates the rule of bit stuffing for fields from the start of frame to CRC delimiter or destroys the fixed form ACK field (two recessive bits) or EOF field (seven recessive bits). As a consequence, all the other stations including the transmitting station detect an error condition and transmit their own error flags. Hence, the resulting sequence of the dominant bits transmitted on the bus consists of the superposition of all the error flags caused by the initial error. The length of this sequence, which constitutes the error flag field of the error frame, can be anywhere between six to twelve bits.

The error delimiter consists of eight recessive bits. After transmitting its error flag each station sends recessive bits and monitors the bus until it sees a recessive bit. A recessive bit at this point indicates that each station has finished sending its error flag. Then each sends seven more recessive bits, which completes the transmission of the error delimiter. The stations on the network can start transmitting only after the three bit intermission field which follows the error delimiter.

Overload Frame

There are two kinds of conditions that lead to the transmission of an overload frame:

- If a receiver requires more time to process data and complete its internal operations before it is ready to receive the next message.
- Detection of a dominant bit during the Intermission field which consists of three recessive bits.

If it is the overload condition 1 that prompts the generation of the overload frame, the transmission of the overload frame must start at the first bit of the Intermission period. On the other hand, overload frames due to the overload condition 2 start right after the detected dominant bit. Similar to the error frame, the overload frame consists of an overload flag with six dominant bits and an overload delimiter with eight recessive bits.

An overload flag destroys the format of the intermission field (three recessive bits). As a consequence, all the other stations detect an overload condition and transmit their own overload flags. Hence, the resulting sequence of the dominant bits transmitted on the bus consists of the superposition of all the overload flags. This sequence, which constitutes the overload flag field of the overload frame, is either six or seven bits long. After transmitting its overload flag each station sends recessive bits and monitors the bus until it sees a recessive bit. A recessive bit at this point indicates that each station

has finished sending its overload flag. Then each sends seven more recessive bits which completes the transmission of the overload delimiter.

At most, two consecutive overload frames can be sent to delay the further transmission of any message.

Intermission Frame

The data frames and the remote frames are separated from the preceding frames by a three bit long Intermission frame. The transmission of neither a data frame nor a remote frame can start before this intermission frame is over. The only action that can be taken during the Intermission frame is the transmission of an overload frame.

Error Detection

For error detection the Controller Area Network implements several mechanisms. Three of these are at the message level and two are at the bit level.

The error detection at the message level is carried out by

- cyclic redundancy checks (CRC)
- frame checks
- acknowledgment error checks

At the bit level the error detection is done by

- bit monitoring
- bit stuffing

The principles of the CRC has been explained in the discussion of the CRC field of the data frame. The CRC check is capable of detecting the following errors:

- Up to 5 single bit errors,
- All odd number of bit errors ,
- All error bursts of length 15 bits or less ,
- A fraction of 0.999939 of the error bursts of length 16 bits,
- A fraction of 0.999969 of the error bursts of length greater than 16 bits.

Frame check is performed by checking the predefined bit fields within a CAN frame. Those are the following:

- CRC delimiter
- ACK delimiter
- end of frame (EOF) bit field
- interframe space

All of these bit fields consist of recessive bits. If a receiver reads a dominant bit in one of these slots a *frame error* (also known as format error) results.

The positive acknowledgment in CAN is implemented via the ACK slot in the data (remote) frame. Upon the successful reception of all the fields up to the ACK field of a message each receiver overwrites the recessive bit in the ACK slot of the transmitted frame. If that bit remains recessive an ACK error is flagged. The possible causes of ACK errors are transmission errors, corrupted ACK fields and the absence of operational receivers in the network.

The transmitting station always monitors the bus to ensure that the transmitted bit level agrees with the actual bit level on the bus. This is called bit monitoring. The transmitter flags a bit error if it does not read the bit it transmitted. This check is obviously not performed for the identifier field and the ACK slot because arbitration and positive acknowledgments overwrite the recessive bits on these fields.

If a receiver, after de-stuffing the received message, detects six or more consecutive bits of the same polarity a bit stuffing error is flagged.

All error detection mechanisms combined, the probability of undetected errors is computed to be 10^{-13}.

Error Confinement

For reliable operation of a network it is important to devise a powerful error detection mechanism. As described in the previous section, CAN has advanced error detection capabilities. However, for increased efficiency and serviceability of the network a mechanism is needed to discriminate between temporary errors and permanent failures. While temporary errors caused by voltage spikes, harsh operation conditions, etc. only reduce the efficiency of the system and are self corrected via re-transmissions, permanent failures, which may be caused by poor connections, faulty cables, defective devices, etc., pose more serious problems in terms of efficiency and maintenance of the network.

CAN utilizes a mechanism called Error Confinement, which is reportedly unique to CAN, to detect permanent failures in the network. Error Confinement is based on the counting of the transmission and reception errors that occur during the network operation. Each node on the network store the error counts in dedicated internal registers called the receive error count register and the transmit error count register. Whenever an error is flagged, the corresponding error count register is incremented. The receive errors have the weight of 1 whereas the transmit errors have the weight of 8 which means that nodes which transmit errors accumulate error counts faster than those that detect received errors. Similarly, any successfully transmitted or received message decrements the corresponding register by 1 (same for both). A high error count indicates a possibly faulty node.

Each node on the network, depending on the count of its internal registers, operates in one of the four modes listed below:

- normal mode
- error active mode
- error passive mode
- bus off mode

Nodes usually operate in the normal mode. In this mode, the error count in both registers is zero. As soon as an error is detected, the error count of the corresponding register goes up and the node enters the error active mode. In this mode, the node

is still fully functional but it is in a state of alert. In case of the detection of an error an error active node transmits an active error flag consisting of six dominant bits (as explained in the discussion of the error frame). The successful reception/transmission of messages decrements the error count registers which causes the node to go back to the normal mode of operation when the count in both registers drop to zero. If, as a result of subsequent detection of errors, the error count in one of the registers exceeds 127 the node enters the so called error passive mode. This represents an increased state of alert for the node. An error passive node, even though still functional, is restricted in how it flags the errors it detects on the bus. Upon detection of an error the error passive node generates a passive error flag consisting of six recessive bits (as opposed to the dominant bits of the active error flag).

There is a good possibility that a node in the error passive node is faulty. The higher the error count, the greater the possibility that the node is faulty. It is possible though for an error passive node to go back to the normal mode of operation. This requires, however, the transmission and reception of a large number of messages without errors.

A node is assumed to be faulty if the error count exceeds 255. When that occurs the node enters the bus off mode, that is, it takes itself off the bus which indicates a permanent failure. That node does not take part in the communication in the network anymore, it does not receive or transmit messages. The other

nodes in the network, on the other hand, continue to communicate with each other. The operation of the network is not affected except the loss of functionality due to the unavailability of the faulty node. According to the ISO 11898 standard a bus off node can go back to the normal mode of operation after having monitored 128 sequences of 11 recessive bits.

Bit Timing and Synchronization

The nominal bit time is the duration of a single bit on the bus. All nodes on a CAN bus are required to have the same nominal bit time for successful communication on the network. The nominal bit time in a CAN bus, as defined in ISO 11898, consists of four non-overlapping time segments.

The synchronization segment (synch-seg) is used to synchronize the nodes on the bus. A bit edge (if there is a signal transition) is expected during this segment. The propagation time segment (prop-seg) is used to compensate for the physical delay times within the network. It is twice the sum of the output driver delay, the propagation time of the signal on the bus and the comparator delay. The phase buffer segment 1 (phase-seg1) and phase buffer segment *2* (phase-seg2) are used to compensate for edge phase errors. Positive phase errors are compensated for by lengthening the phase-seg1, while negative phase errors are compensated for by shortening the phase-seg2. The transmitted bit is sampled by the receiving nodes at the end of the phase buffer segment 1.

CAN protocol does not use a dedicated clock signal to serve as a timing reference. Therefore, the synchronized communication between the nodes on the CAN network is accomplished by two methods: hard synchronization and re-synchronization. Both type of synchronization occurs in the CAN controller of each receiving node.

Hard synchronization occurs when a falling edge (1 → 0 or recessive → dominant) is detected during the bus-idle time. The signal transition indicates the start of a new message frame (transmission of the SOF bit). Hard synchronization initializes the bit timing logic of each controller and ensures that the SOF bit edge lies in the synchronization segment of the nominal bit time at each receiver.

Even though hard synchronization would be sufficient in theory, inconsistencies in the system such as small differences in receiver and transmitter clocks, oscillator drift, or inherent delays in the network cause synchronization problems. To remedy this problem re-synchronization is needed. There are two possible scenarios; either the bit edge will arrive late (after the synch-seg) or early (next bits bit edge falls within phase-seg2). In the former case, which is defined as a positive phase error, the phase-seg1 is lengthened, while in the latter case, which is defined as a negative phase error, the phase-seg2 is shortened. The amount by which the bit time is shortened or lengthened is determined by a user programmable synchronization jump width (SJW).

From the discussion above it is obvious that a certain amount of bit edges (signal transitions) is necessary to maintain synchronization throughout the transmission. The predefined bits (CRC delimiter, ACK delimiter, etc.) and the bit stuffing rule ensures that the required minimum number of edges are present in the transmitted bit stream.

APPENDIX B
A TWO-STATION MARKOV MODEL WITH OUTPUT DELAY

As demonstrated in Chapter 4, one of the methods used to control the pace of the production is to make adjustments to the capacity of the input buffer. If the excess accumulation of parts in its buffer causes a station to operate inefficiently, it might decide to reduce its buffer capacity even if this slows down production. On the other hand, if a station can tolerate excess inventory build-up due to increased production, it might decide to increase its buffer capacity. In general, that is, ignoring the interference caused by other stations, an increase in buffer capacity leads to an increase in production and in inventory cost. Simulations show that in many cases increasing the buffer capacity even by one, for example from zero to one, might result in large fluctuations in profit. Therefore, it is worthwhile to investigate whether it is possible to make a finer adjustment to buffer capacity than moving it up/down by one.

For the sake of simplicity, a two-station arrangement, as shown in Figure B-1, is used for modeling purposes. It consists of two stations with a finite-size buffer in the middle. The only control parameter in this configuration is the buffer size. Figure B-2 shows a hypothetical plot of profit with respect to buffer size for the two-station arrangement. The circles indicate

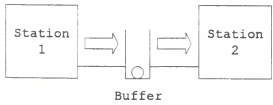


Figure B-1. Two-station arrangement.

the discrete points, for which profit is available. Since the buffer size can be assigned integer values only, the performance of the system is determined only for these discrete values. In the adaptive policy *buffer_size*, for example, the system alternates between states corresponding to discrete buffer size values. In some cases, considerable profit is lost due to this

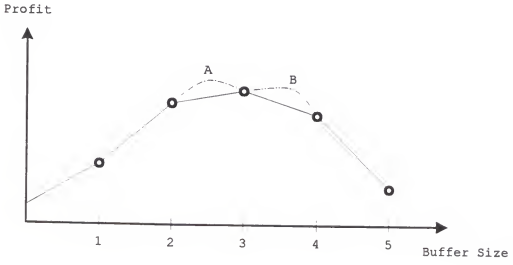


Figure B-2. Hypothetical plot of profit for two-station model.

wandering around. If however, a method can be devised to define the operation of the system between two discrete points, a higher profit might be achieved. In addition, this would help eliminate the high profit losses that occur due to the continuous search method (oscillatory behavior) employed by the adaptive policy.

According to the plot in Figure B-2, the maximum profit is achieved when buffer size is set to three. It is not clear however, how profitable the system would be between buffer size values two and three or between three and four. It is quite possible for the profit plot to follow the path designated as A or the path designated as B in Figure B-2. In either case, at some point along the path, the system would yield a higher profit than the current maximum, which corresponds to buffer size value three.

A candidate approach to achieve a continuous function of profit with respect to buffer size is investigated here. The proposed method involves selective delays to part transfer from station 1 into the intermediate buffer. Figure B-3 depicts the rules for part transfer in the proposed method for a two-station system with buffer capacity four. As shown in the figure, shipment of parts occurs without any delay if the buffer is empty (Figure B-3a) or the buffer level is at one (Figure B-3b) or two (Figure B-3c). However, in Figure B-3d, there are already three parts waiting in the intermediate buffer. When the fourth part becomes available, it is not placed into the intermediate buffer

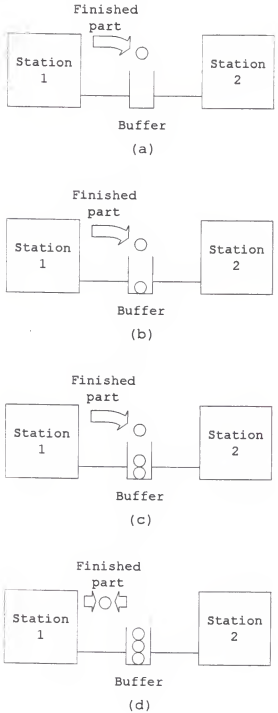


Figure B-3. Shipment of a finished part into the intermediate buffer with a capacity of four when
a) Buffer is empty (no delay);
b) One part is already in the buffer (no delay);
c) Two parts are already in the buffer (no delay);
d) Three parts are already in the buffer (delay).

immediately. It waits at the output of station 1 for a certain amount of time and then proceeds to the buffer. For modeling purposes, the output delay is represented with the release probability P_R (or holding probability P_H). In other words, when station 1 finishes a part, the probability of the part's release, i.e. its shipment into the intermediate buffer, is determined by $P_R = 1-P_H$. A release probability of zero ($P_R=0$) indicates that the part is never released and the buffer capacity is therefore exactly three. A release probability of one ($P_R=1$) implies that the finished part is sent to the buffer without any delay. Consequently, the buffer capacity is exactly four. For $0<P_R<1$, the shipment of part is delayed, hence the buffer capacity remains at three during the delay. As soon as the part is released however the capacity increases to four. In a sense, the buffer capacity is three with a probability of P_H, and four with a probability of $1-P_H$. Thus, it can be argued that for $P_H=P_R=0.5$, the buffer capacity is effectively 3.5. To avoid confusion and to keep the notation simple, only the holding probability will be used in the derivation below and it will be referred to as H rather than P_H.

As stated before, the objective of this analysis is to investigate whether it is possible to achieve a higher profit level employing an alternative approach to using discrete buffer size values. The approach described above provides a continuous function of profit with respect to buffer size. Hence, the analysis of this approach will show whether the maximum of the

profit function occurs only at discrete, integer values of buffer size. More specifically, the question asked is whether, in a two-station arrangement with an intermediate buffer, the buffer capacity 3.5 can yield a higher profit than the buffer capacities three and four.

Figure B-4 shows the state diagram of the Markov model developed to analyze the system described above. In the model, M is the buffer size, H is the holding probability of station 1, and w_n represents the process rate of n'th station. Table B-1 gives a description of all the possible states in a two-station, one intermediate buffer Markov model with holding at the output. In the state diagram, P_S represents the probability of state WS. Similarly, P_n represents the probability of state n, where n indicates the buffer level. It should be noted that there are two states, WW and HW, at buffer level M-1. There are also two states, WW and BW at buffer level M. Therefore,

$$\begin{aligned} P_{M-1} &= \underset{(WW)}{P_{M-1}} + \underset{(HW)}{P_{M-1}} \\ P_M &= \underset{(WW)}{P_M} + \underset{(BW)}{P_M} \end{aligned} \qquad \text{(B-1)}$$

It should be emphasized that holding can occur only if the buffer level is M-1, which is represented by state HW and probability $\underset{(HW)}{P_{M-1}}$ in the model. For all buffer levels less than M-1, the finished part is released immediately causing a transition to one of WW states.

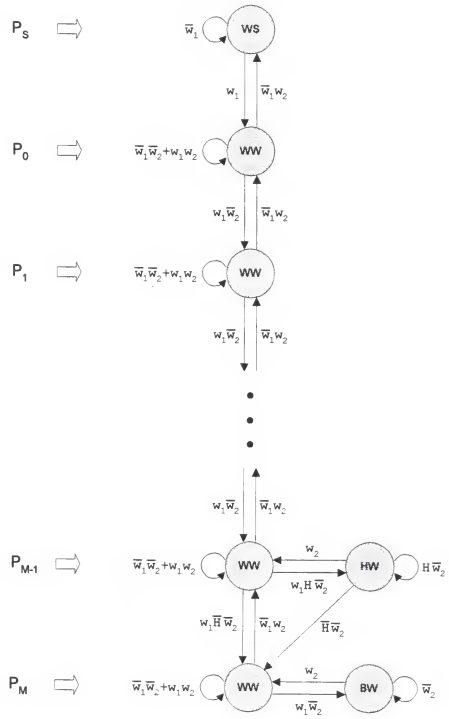


Figure B-4. State diagram of the Markov model for a two-station arrangement with intermediate buffer.

Table B-1. Description of all the states in the Markov model.

State	Station 1	Station 2	Buffer Level
WS	Working	Starved	0
WW	Working	Working	0,1,..,M
HW	Holding	Working	M-1
BW	Blocked	Working	M

The first step in the analysis is the derivation of the inventory cost. The work-in-process (WIP) inventory of the system is given by

$$WIP = P_1 + 2P_2 + \ldots + (M-1)\underset{(WW)}{P_{M-1}} + M\underset{(HW)}{P_{M-1}} + M\underset{(WW)}{P_M} + M\underset{(BW)}{P_M} \qquad \text{(B-2)}$$

As indicated in Equation B-3, the state probabilities sum to one. Multiplying both sides by M and re-arranging the equation

$$P_S + P_0 + P_1 + P_2 + \ldots + \underset{(WW)}{P_{M-1}} + \underset{(HW)}{P_{M-1}} + \underset{(WW)}{P_M} + \underset{(BW)}{P_M} = 1 \qquad \text{(B-3)}$$

the following equation is obtained:

$$M(1 - P_S - P_0) = M\left(P_1 + P_2 + \ldots + \underset{(WW)}{P_{M-1}} + \underset{(HW)}{P_{M-1}} + \underset{(WW)}{P_M} + \underset{(BW)}{P_M}\right) \qquad \text{(B-4)}$$

or

$$M(1 - P_S - P_0) = \left(P_1 + 2P_2 + \ldots + (M-1)\underset{(WW)}{P_{M-1}} + M\underset{(HW)}{P_{M-1}} + M\underset{(WW)}{P_M} + M\underset{(BW)}{P_M}\right) + \left((M-1)P_1 + (M-2)P_2 + \ldots + \underset{(WW)}{P_{M-1}}\right) \qquad \text{(B-5)}$$

The first term on the left side in the equation above is the same as the WIP term in Equation B-2. Hence Equation B-2 can be re-written as

$$WIP = M\left(1 - P_S - P_0\right) - \left(\left(M - 1\right)P_1 + \left(M - 2\right)P_2 + \ldots + P_{\substack{M-1 \\ (WW)}}\right) \qquad (B\text{-}6)$$

In steady state, the frequency of transitions from state WS to state WW (empty buffer) is the same as the number of transitions from state WW (empty buffer) to state WS. Thus,

$$P_S\omega_1 = P_0\bar{\omega}_1\omega_2 \quad \Rightarrow \quad P_S = \frac{\bar{\omega}_1\omega_2}{\omega_1}P_0 = k_0P_0 \qquad (B\text{-}7)$$

Similarly,

$$\begin{aligned} P_1 &= \frac{\omega_1\bar{\omega}_2}{\bar{\omega}_1\omega_2}P_0 = k_1P_0 \\ P_2 &= \frac{\omega_1\bar{\omega}_2}{\bar{\omega}_1\omega_2}P_1 = k_1^2P_0 \\ &\vdots \\ P_{\substack{M-1 \\ (WW)}} &= k_1^{(M-1)}P_0 \end{aligned} \qquad (B\text{-}8)$$

As indicated by Equation B-8, all state probabilities except $P_{\substack{M-1 \\ (WW)}}$, $P_{\substack{M \\ (WW)}}$ and $P_{\substack{M \\ (BW)}}$ are written in terms of P_0. Therefore, it is possible to express WIP solely in terms of P_0.

$$WIP = M\left(1 - k_0P_0 - P_0\right) - \left(\left(M - 1\right)k_1 + \left(M - 2\right)k_1^2 + \ldots + 2k_1^{(M-2)} + k_1^{(M-1)}\right)P_0 \qquad (B\text{-}9)$$

Simplifying Equation B-9,

$$WIP = M - M\left(1 + k_0\right) P_0 - S P_0 \tag{B-10}$$

where

$$S = \left(\frac{k_1}{k_1 - 1}\right)\left[\frac{k_1}{k_1 - 1}\left(k_1^{(M-1)} - 1\right) - (M - 1)\right] \tag{B-11}$$

In the equation above, WIP is expressed in terms of buffer size M, process parameters w_1 and w_2, and state probability P_0. Note that P_0 is the only state probability present in the WIP equation.

The inventory cost C can easily be found by

$$C = (WIP)\, h = \left[M - M\left(1 + k_0\right) P_0 - S P_0\right] h \tag{B-12}$$

where h is the inventory holding cost per unit per time. It might seem interesting that the holding probability H does not appear directly in the inventory cost. It is embedded in state probability P_0. In other words, P_0 is a function of H.

$$P_0 = f(H) \tag{B-13}$$

Thus, varying the holding probability H affects the value of P_0, which in turn changes the inventory cost C. It necessary to find the relationship between C and H. An increase in holding probability implies a longer output delay which means that the system spends more time in HW state. Consequently, the

probability $P_{(BW)}^{M-1}$ goes up with increasing H. The node equation at node $P_{(WW)}^{M}$ yields

$$P_{(WW)}^{M}\,\bar{\omega}_1\omega_2 = P_{(WW)}^{M-1}\,\omega_1\bar{\omega}_2 - P_{(BW)}^{M-1}\,\omega_2 \qquad \text{(B-14)}$$

Equation B-14 suggests that in order $P_{(WW)}^{M}$ to increase $P_{(WW)}^{M-1}$ must increase as well. However, a close look to Equations B-8 and B-15 reveal that it is not possible for both probabilities to increase at the same time. It is evident from (B-8) that an increase in $P_{(WW)}^{M-1}$ means an increase in state probabilities P_S, P_0, P_1,..., P_{M-2}. Likewise, Equation B-15, which gives the number of transitions form $P_{(WW)}^{M}$ to $P_{(BW)}^{M}$ and vice versa, indicates that an increase in $P_{(WW)}^{M}$ causes a proportional increase in $P_{(BW)}^{M}$. This results in the

$$P_{(BW)}^{M}\,\omega_2 = P_{(WW)}^{M}\,\omega_1\bar{\omega}_2 \quad \Rightarrow \quad P_{(BW)}^{M} = P_{(WW)}^{M}\,\frac{\omega_1\bar{\omega}_2}{\omega_2} \qquad \text{(B-15)}$$

increase of all state probabilities, which is not possible according to Equation B-3. Therefore, the only conclusion that can be drawn is that $P_{(WW)}^{M}$ decreases with increasing holding probability H. As a result, it can be shown that

$$\frac{d\left(P_{(WW)}^{M} + P_{(BW)}^{M}\right)}{dH} < 0 \quad \Rightarrow \quad \frac{dC}{dH} < 0 \qquad \text{(B-16)}$$

More specifically, from (B-12)

$$\frac{dC}{dH} = \frac{dC}{dP_0}\frac{dP_0}{dH} = -\left(M\left(1 - k_0\right) + S\right) h \left(\frac{dP_0}{dH}\right) \qquad \text{(B-17)}$$

The second phase of the analysis is the derivation of the revenue. The only state, at which the second station is inactive, is state WS. Therefore, the production rate of the system is given by

$$\text{Throughput} = \left(1 - P_0\right)\omega_2 = \left(1 - k_0 P_0\right)\omega_2 \qquad \text{(B-18)}$$

Since a reward of r is collected for each produced part, the revenue R turns out to be

$$R = (\text{Throughput})\, r = \left(1 - k_0 P_0\right)\omega_2 r \qquad \text{(B-19)}$$

Revenue is related to H as follows:

$$\frac{dR}{dH} = \frac{dR}{dP_0}\frac{dP_0}{dH} = -\left(k_0 \omega_2 r\right)\left(\frac{dP_0}{dH}\right) \qquad \text{(B-20)}$$

Hence,

$$\frac{\frac{dR}{dH}}{\frac{dC}{dH}} = \frac{k_0 \omega_2 r}{\left(M\left(1 - k_0\right) - S\right) h} = F \qquad \text{(B-21)}$$

An interesting observation about this equation is that it depends neither on holding probability H, nor any state probability. When coupled with (B-16), the lack of dependence on H implies a monotonic increase or decrease of profit with H. Monotonic nature

of the profit function comes from the fact that F is a constant that depends only on process parameters w_1 and w_2, buffer size M, and the ratio of unit reward (r) to unit cost (h). The sign of F is determined by those parameters and is not affected by changes in H. In other words, at any H value, a small change ΔH causes either always an increase or always a decrease in profit. From (B-21),

$$\frac{dProfit}{dH} = \frac{dR}{dH} - \frac{dC}{dH} = \frac{dC}{dH}(F - 1) \qquad \text{(B-22)}$$

Combining (B-22) with (B-16), the following can be said about the change of profit with H:

$$\begin{aligned} &If \quad F > 1 \Rightarrow \frac{dProfit}{dH} < 0 \\ &If \quad F = 1 \Rightarrow \frac{dProfit}{dH} = 0 \\ &If \quad F < 1 \Rightarrow \frac{dProfit}{dH} > 0 \end{aligned} \qquad \text{(B-23)}$$

Consequently, the direction of change in profit remains the same as H goes from zero to one. As shown in (B-23), profit either monotonically increases (F<1), or monotonically decreases (F>1), or remains the same (F=1). As a result, it can never exceed the maximum of either H=0 or H=1. In other words, maximum profit occurs only at discrete, integer values of buffer size.

APPENDIX C
ASSEMBLY AND C SOURCE CODE USED IN THE IMPLEMENTATION

The scanner module is in charge of controlling the network, stations are in charge of production. Since these are completely different tasks, separate programs have to be written for both type of modules.

Source Code for Station Modules

Since all stations operate the same way, they run the same programs. This program consists of the following C and assembly modules:

- Code.c - contains the main program.
- CanMessage.c - contains code for CAN message transmission.
- Canisr.c - contains interrupt service routines.
- Learn.c - implements the learn strategy.
- S0.c - contains the string output routines.
- Timer01.c - implements timer interrupt for time base.
- RandomParam.c - contains the random number generator.
- Can.asm - contains assembly code for CAN setup

Code.c is the main program module. One of its tasks is the initialization of the station module. It also calls routines in *Can.asm*, which prepare the station for CAN communication. *Code.c*

contains a loop in which the processor continuously checks whether the next clock tick has occurred. When the program detects a new clock tick, it checks the current status and determines whether it is time for a new event, such as a breakdown, repair, or completion of a part. In case of a new event, the program changes the status of the station and updates the parameters. To get new parameter values such as process time, repair time, and breakdown time, a call is placed to *RandomParam.c*, which contains the random number generator.

Since message exchange between stations is interrupt driven, the arrival of a CAN message triggers an interrupt. The code in *Code.c* also checks for interrupts (CheckRequest, CheckPermission, CheckShipment routines) to see if a message has arrived. The interrupt service routines for production family of messages are coded in *Canisr.c*.

After the completion of an interval, each station waits (WaitDataRequest routine) until it receives a request (Request_Stats message) to send data pertaining to its performance in the past interval. After the data is transmitted (TransmitStats routine) the program calls *Learn.c* to fine-tune its control strategy.

```
/* ------------ Reads166 generated header ------------
 * module description : main program
 * module file name   : Code.C
 * ---------------------------------------------------- */

// for 10ms clock 0x9E58 should be used for reload value.
// calculation:  10000h-25000
// reload value for overflows every 10 millisecond.
// with CPU clock=20MHz (50 nanosecond processor clock) and the
// prescalar set to 8.
// That is, 8*50*25000=10,000,000 nanoseconds or 10 milliseconds.

#define MSEC_COUNT  0x9E58

#define MAXBUFFERCAPACITY   10
#define INTERVALLENGTH   10000
#define UNITREWARD         250
#define INVENTORYUNITCOST    1
#define HIGHER               1
#define LOWER               -1

// --- global variables ---
predef unsigned int T0, T0REL, T01CON, T0IC, S0RIC;
predef unsigned int CTLST, INTREG;
predef unsigned int MSGCTL1, MSGCTL2, MSGCTL3, MSGCTL7,
MSG_CTL10;
predef unsigned bit T0R, S0RIR, IEN;
predef unsigned bit DP2_0, P2_0;

// Time base
unsigned int MSEC;              // milliseconds, low word
unsigned int SECONDS;           // seconds
unsigned int uSeconds, uMsec;

// CAN (Messaging) variables
unsigned int uCanNodeID, uLastNodeID;
unsigned int uRequestFlag, uShipmentFlag, uPermissionFlag;
unsigned int nMessageType;

// Production parameters
unsigned int uTimeToFinish, uServiceTime;
unsigned int uTimeToBreakDown, uBreakDownTime;
unsigned int uTimeToRepair, uRepairTime;

// Line & Node variables
unsigned int uStatus;
unsigned int uOutput, uOldOutput, uLineOutput;
unsigned int uInputBuffer, uInputBufferSize, uOldInputBufferSize;
unsigned int uInvUnitCost;
float fInvLevel, fOldInvLevel;
long nProfit, nOldProfit, nTotalProfit;
long nReward, nRevenue, nOldRevenue;
```

```
long nInv, nInvCost, nOldInvCost;

// Learn variables
unsigned int uDirBufferSize, uDirProfit, uDirOutput,
uDirInvLevel;

// Random Number parameters
unsigned int uServiceLow, uRepairLow, uBreakDownLow;
unsigned int uServiceHigh, uRepairHigh, uBreakDownHigh;
long nSeed;

long count;


// --- constants ---
enum uStatus {STARVING=1,WORKING,BLOCKED,BLOCKED_DOWN,DOWN};
enum nMessageType
{REQUEST=1,SHIPMENT,PERMISSION,CANNODEID,STATS};


// --- Assembly Functions ---
void can_init(void);
void can_setup_messaging(unsigned int);
void wait_rmt_rqst(unsigned int);
void wait_msg(unsigned int);
void clr_newdat(unsigned int);
void check_data(unsigned int);
void set_field_nodeid(unsigned int);
void set_field_prodcount(unsigned int);
void set_field_invcost(long);
void set_field_buffersize(unsigned int);
unsigned int get_field_nodeid(unsigned int);


// --- C Functions ---
void InitializeVars(void);
void ResetStats(void);
void SendMessage(unsigned int);
void ReceiveLastNodeID();
void Delay(unsigned int);
void ReportStatus(void);
void InputCanNodeID(void);
void WaitDataRequest(void);
void CompileStats(void);
void TransmitStats(void);
void PrintStats(void);
void EvaluatePerformance(void);
void Learn(void);
```

```
// main function
main(void){

 S0RIC=0;
 InitializeVars();
 can_init();
 IEN=1;                          // enable interrupts

 InputCanNodeID();
 TransmitNodeID();
 ReceiveLastNodeID();

 SendStr("\n Last ID received ..");
 SendInt(uLastNodeID);
 SendStr("\n");

 can_setup_messaging(uCanNodeID);

 SendStr("\nEmulation Starting...");

 if (uCanNodeID==1) {
   nReward=250;
   uInputBufferSize=10;
   uServiceLow=30;
   uServiceHigh=50;
   uRepairLow=160;
   uRepairHigh=240;
   uBreakDownLow=600;
   uBreakDownHigh=1000;
   uStatus=WORKING;
   uTimeToBreakDown=GetNewBreakDownTime();
   uTimeToFinish=GetNewServiceTime();
 }
 else if (uCanNodeID==2) {
   nReward=250;
   uInputBufferSize=1;
   uServiceLow=30;
   uServiceHigh=50;
   uRepairLow=160;
   uRepairHigh=240;
   uBreakDownLow=600;
   uBreakDownHigh=1000;
   uStatus=STARVING;
   uTimeToBreakDown=GetNewBreakDownTime();
 }
 else if (uCanNodeID==3) {
   nReward=250;
   uInputBufferSize=1;
   uServiceLow=30;
   uServiceHigh=50;
   uRepairLow=160;
   uRepairHigh=240;
```

```
  uBreakDownLow=600;
  uBreakDownHigh=1000;
  uStatus=STARVING;
  uTimeToBreakDown=GetNewBreakDownTime();
}
else if (uCanNodeID==4) {
  nReward=250;
  uInputBufferSize=1;
  uServiceLow=30;
  uServiceHigh=50;
  uRepairLow=160;
  uRepairHigh=240;
  uBreakDownLow=600;
  uBreakDownHigh=1000;
  uStatus=STARVING;
  uTimeToBreakDown=GetNewBreakDownTime();
}

DP2_0=0;          // configure Port 2_0 as input

do {

  ResetStats();
  while (!P2_0);
  T0init(MSEC_COUNT);           // initialize Timer0

  count=0;
  do {
    count++;
    if (uCanNodeID==1) {
      CheckPermission();
    }
    else if (uCanNodeID==uLastNodeID) {
      CheckRequest();
      CheckShipment();
    }
    else {
      CheckRequest();
      CheckPermission();
      CheckShipment();
    }

    if(SECONDS!=uSeconds) {
      uSeconds=SECONDS;
    }

    if(MSEC!=uMsec) {          // new clock tick
      uMsec=MSEC;
      nInv+=uInputBuffer;

      if(uStatus==WORKING) {
        uTimeToFinish--;
```

```
    uTimeToBreakDown--;
    if(uTimeToFinish==0) {
      if(uCanNodeID==uLastNodeID) {
        uOutput++;
        uLineOutput++;
        if (uTimeToBreakDown==0) {
          uStatus=DOWN;
          uTimeToRepair=GetNewRepairTime();

        }
        else {
          if (uInputBuffer==0) {
            uStatus=STARVING;
          }
          else {
            uStatus=WORKING;
            uInputBuffer--;
            uTimeToFinish=GetNewServiceTime();
          }
        }
      }
      else {
        SendMessage(REQUEST);
        if (uTimeToBreakDown==0) {
          uStatus=BLOCKED_DOWN;
          uTimeToRepair=GetNewRepairTime();
        }
        else {
          uStatus=BLOCKED;
        }
      }
    }
    else {
      if (uTimeToBreakDown==0) {
        uStatus=DOWN;
        uTimeToRepair=GetNewRepairTime();
      }
    }
  }
  else if (uStatus==DOWN) {
    uTimeToRepair--;
    if (uTimeToRepair==0) {
      uTimeToBreakDown=GetNewBreakDownTime();
      if (uTimeToFinish==0) {
        if (uCanNodeID==1) {
          uStatus=WORKING;
          uTimeToFinish=GetNewServiceTime();
        }
        else {
          if (uInputBuffer==0) {
            uStatus=STARVING;
          }
```

```
            else {
              uStatus=WORKING;
              uInputBuffer--;
              uTimeToFinish=GetNewServiceTime();
            }
          }
        }
        else {
          uStatus=WORKING;
        }
      }
    }
    else if (uStatus==BLOCKED_DOWN) {
      uTimeToRepair--;
      if (uTimeToRepair==0) {
        uStatus=BLOCKED;
        uTimeToBreakDown=GetNewBreakDownTime();
      }
    }
  }
 } while(P2_0);

 T0IC=0;
 CompileStats();
 WaitDataRequest();
 TransmitStats();
 PrintStats();
 Learn();

 SendStr("\n End interval");
} while(1);

}


void CheckRequest() {
  if(uRequestFlag) {
    if(IsPermissionGranted()) {
      uRequestFlag=0;
      SendMessage(PERMISSION);
    }
  }
}


void CheckPermission() {
  if(uPermissionFlag) {
    uPermissionFlag=0;
    SendMessage(SHIPMENT);
    uOutput++;
    if (uStatus==BLOCKED_DOWN) {
      uStatus=DOWN;
```

```
    }
    else if (uStatus==BLOCKED) {
      if (uCanNodeID==1) {
        uStatus=WORKING;
        uTimeToFinish=GetNewServiceTime();
      }
      else {
        if(uInputBuffer==0) {
          uStatus=STARVING;
        }
        else {
          uStatus=WORKING;
          uInputBuffer--;
          uTimeToFinish=GetNewServiceTime();
        }
      }
    }
  }
}


void CheckShipment() {
  if(uShipmentFlag) {
    uShipmentFlag=0;
    uInputBuffer++;
    if(uStatus==STARVING) {
      uStatus=WORKING;
      uTimeToFinish=GetNewServiceTime();
      uInputBuffer--;
    }
  }
}


unsigned int IsPermissionGranted() {
 if (uStatus==BLOCKED) {
   return(0);
 }
 if (uStatus==BLOCKED_DOWN) {
   return(0);
 }
 if (uStatus==DOWN) {
   return(0);
 }
 if (uStatus==STARVING) {
   return(1);
 }
 if (uStatus==WORKING) {
   if (uInputBuffer>=uInputBufferSize) {
     return(0);
   }
   else {
```

```
    return(1);
   }
  }
}


void TransmitNodeID() {
  wait_rmt_rqst(7);                 // wait for remote request
                                    // asking for Node ID
  set_field_nodeid(uCanNodeID);     // write Node ID into the data
                                    // byte 1
  Delay(uCanNodeID);
  SendMessage(CANNODEID);           // transmit Node ID
}


void ReceiveLastNodeID() {
  wait_msg(8);
  uLastNodeID=get_field_nodeid(8);
  clr_newdat(8);
}


void InputCanNodeID() {
  SendStr("\nInput CAN Node Number (1..9) : ");
  while(!PeekChar());           // wait for a char
  uCanNodeID=GetChar()-'0';     // get node number
  uCanNodeID%=10;               // minimal error detection
  SendInt(uCanNodeID);          // echo node number
}


void Delay(unsigned int uID) {      // causes a small delay
  T0init(MSEC_COUNT);               // duration: uID*200 millisec
  while(MSEC < 20*uID);
  T0IC=0;
}


void WaitDataRequest(void) {    // receives CAN message
unsigned int ID;                // Request_Stats from scanner

  do {                          // loop
    wait_msg(9);                // wait until message arrives
    SendStr("\nreq rec..");     // echo the arrival
    ID=get_field_nodeid(9);     // extract node ID from data
    SendStr(" ID=");            // print ID #
    SendInt(ID);
    clr_newdat(9);              // reset
  } while (ID!=uCanNodeID);     // if ID# doesn't match go to loop
}
```

```
void CompileStats(void) {      // find out revenue, cost etc.
  fInvLevel=(1.0*nInv)/INTERVALLENGTH;
  nRevenue=nReward*uOutput;
  nInvCost=nInv*uInvUnitCost;
  nProfit=nRevenue-nInvCost;
}


// sends stats in transmit object 10
void TransmitStats(void) {           // transmit stats to scanner
  set_field_prodcount(uOutput);       // copy production count
  set_field_invcost(nInvCost);        // copy inventory cost
  set_field_buffersize(uInputBufferSize); // copy buffersize
  SendMessage(STATS);                 // call transmit routine
  SendStr("\n stats transmitted");    // acknowledge
}


void PrintStats(void) {              // print production data
  SendStr("\n Stats : PC ");         // on the screen
  SendInt(uOutput);
  SendStr(" Inv ");
  SendLongInt(nInv);
  SendStr(" InvC ");
  SendLongInt(nInvCost);
  SendStr(" Rev ");
  SendLongInt(nRevenue);
  SendStr(" Pr ");
  SendLongInt(nProfit);
  SendStr(" BS ");
  SendInt(uInputBufferSize);
  SendStr(" done\n");
}


void ResetStats(void) {
  uSeconds=0;
  uMsec=0;

  fOldInvLevel=fInvLevel;
  uOldInputBufferSize=uInputBufferSize;
  uOldOutput=uOutput;
  nOldRevenue=nRevenue;
  nOldInvCost=nInvCost;
  nOldProfit=nProfit;

  fInvLevel=0;

  nInv=uInputBuffer;
  uOutput=0;
  nRevenue=0;
```

```
  nProfit=0;
  nInvCost=0;
}

void InitializeVars() {

// Random
  nSeed=1234;

// Time base
  uSeconds=0;
  uMsec=0;

// Production
  uLineOutput=0;
  uInputBuffer=0;

// Messaging
  uRequestFlag=0;
  uShipmentFlag=0;
  uPermissionFlag=0;
  uLastNodeID=0;

// Control & Learn
  uInvUnitCost=INVENTORYUNITCOST;
  nInvCost=0;
  nRevenue=0;
  nProfit=0;
  nTotalProfit=0;
  uDirBufferSize=LOWER;
}
```

```
/* ------------ Reads166 generated header ------------
 * module description : routines for CAN message transmission
 * module file name   : CanMessage.c
 * ---------------------------------------------------- */

void out_msg(unsigned int);
unsigned int TransmissionError(void);



void SendMessage(unsigned int uMessageType) {

  if (uMessageType==REQUEST) {              // REQUEST -> obj 4
    TransmitObj(4);
  }
  else if (uMessageType==SHIPMENT) {        // SHIPMENT -> obj 5
    TransmitObj(5);
  }
  else if (uMessageType==PERMISSION) {      // PERMISSION -> obj 6
    TransmitObj(6);
  }
  else if (uMessageType==CANNODEID) {       // CANNODEID -> obj 7
    TransmitObj(7);
  }
  else if (uMessageType==STATS) {
    TransmitObj(10);
  }
}



void TransmitObj(unsigned int nObjNo) {
int nTransmissionCount=0;
  do {
    out_msg(nObjNo);                        // call assembly routine
in Rcanlib.inc
    nTransmissionCount++;
  } while(TransmissionError() && nTransmissionCount<3);
  if (nTransmissionCount>=3) {
    SendStr("\n\n TRANSMISSION ERROR !!!  (3 times) \n");
  }
}



unsigned int TransmissionError(void) {
unsigned int uErrorStatus;
  uErrorStatus=CTLST&&0x0700;
  if (uErrorStatus==3) {
    SendStr("\n ERROR: NAck \n");
  }
  else if (uErrorStatus==4) {
```

```
    SendStr("\n ERROR: Bit1 \n");
  }
  else if (uErrorStatus==5) {
    SendStr("\n ERROR: Bit0 \n");
  }
  return(uErrorStatus);
}
```

```
/* ------------ Reads166 generated header ------------
 * module description : Interrupt service routines for
 *                      production family CAN messages
 * module file name   : Canisr.C
 * ---------------------------------------------------- */

predef unsigned int INTREG;

void interrupt(0x100) CANIsr(void){
unsigned int nIntID;

  nIntID=INTREG&0x00FF;       // read CAN Interrupt Register to
find out
                              // the interrupt source

  if (nIntID==3) {            // REQUEST Received
    uRequestFlag=1;           // set request flag
    MSGCTL1=0xFFFD;           // clear INTPND of message control
                              // register of obj 1
  }
  else if (nIntID==4) {       // SHIPMENT RECEIVED
    uShipmentFlag=1;          // set shipment flag
    MSGCTL2=0xFFFD;           // clear INTPND of message control
                              // register of obj 2
  }
  else if (nIntID==5) {       // PERMISSION RECEIVED
    uPermissionFlag=1;        // set permission flag
    MSGCTL3=0xFFFD;           // clear INTPND of message control
                              // register of obj 3
  }
}
```

```
/* ------------ Reads166 generated header ------------
 * module description : Learn module
 * module file name   : Learn.c
 * ------------------------------------------------- */


void Learn(void) {

 DetermineDir();

// Rule 1
 if (uDirBufferSize==HIGHER && uDirProfit==HIGHER) {
   uDirBufferSize=HIGHER;
   IncreaseInputBufferSize();
   SendStr("\n*** Rule 1 ***\n");
   return;
 }
// Rule 2
 if (uDirBufferSize==LOWER && uDirProfit==HIGHER) {
   uDirBufferSize=LOWER;
   ReduceInputBufferSize();
   SendStr("\n*** Rule 2 ***\n");
   return;
 }
// Rule 3a
 if (uDirBufferSize==HIGHER && uDirProfit==LOWER &&
uDirOutput==HIGHER && uDirInvLevel==HIGHER) {
   uDirBufferSize=LOWER;
   ReduceInputBufferSize();
   SendStr("\n*** Rule 3A ***\n");
   return;
 }
// Rule 3b
 if (uDirBufferSize==HIGHER && uDirProfit==LOWER &&
uDirOutput==LOWER) {
   uDirBufferSize=HIGHER;
   IncreaseInputBufferSize();
   SendStr("\n*** Rule 3B ***\n");
   return;
 }

// Rule 4a
 if(uDirBufferSize==LOWER && uDirProfit==LOWER &&
uDirInvLevel==HIGHER) {
   uDirBufferSize=LOWER;
   ReduceInputBufferSize();
   SendStr("\n*** Rule 4A ***\n");
   return;
 }
```

```
// Rule 4b
 if(uDirBufferSize==LOWER && uDirProfit==LOWER &&
uDirInvLevel==LOWER && uDirOutput==LOWER) {
   uDirBufferSize=HIGHER;
   IncreaseInputBufferSize();
   SendStr("\n*** Rule 4B ***\n");
   return;
 }
}



void IncreaseInputBufferSize(void) {
 if (uInputBufferSize<MAXBUFFERCAPACITY)  uInputBufferSize++;
}



void ReduceInputBufferSize(void) {
 if (uInputBufferSize>0) uInputBufferSize--;
}



void DetermineDir(void) {

 if (nProfit>=nOldProfit) {
   uDirProfit=HIGHER;
 }
 else {
   uDirProfit=LOWER;
 }

 if (uOutput>=uOldOutput) {
   uDirOutput=HIGHER;
 }
 else {
   uDirOutput=LOWER;
 }

 if (fInvLevel>fOldInvLevel) {
   uDirInvLevel=HIGHER;
 }
 else {
   uDirInvLevel=LOWER;
 }
}
```

```
/* ------------ Reads166 generated header ------------
 * module description : string output routines
 * module file name   : S0.c
 * ---------------------------------------------------- */

predef unsigned int S0TBUF, S0TIC;
predef bit S0TIR;
predef unsigned int S0RBUF, S0RIC;
predef bit S0RIR;


#pragma DebugOff
void SendLongInt(long n){        // prints a long integer (4-byte
long nNum, nRemain, nResult;     // integer) on the screen of
                                 // host PC.
 if(n & 0x80000000)
  {
   SendChar('-');
   nNum=(~n)+1;
   }
  else nNum=n;

 nResult=nNum/1000000000;
 nRemain=nNum-nResult*1000000000;
 SendChar(nResult+'0');
 nNum=nRemain;

 nResult=nNum/100000000;
 nRemain=nNum-nResult*100000000;
 SendChar(nResult+'0');
 nNum=nRemain;

 nResult=nNum/10000000;
 nRemain=nNum-nResult*10000000;
 SendChar(nResult+'0');
 nNum=nRemain;

 nResult=nNum/1000000;
 nRemain=nNum-nResult*1000000;
 SendChar(nResult+'0');
 nNum=nRemain;

 nResult=nNum/100000;
 nRemain=nNum-nResult*100000;
 SendChar(nResult+'0');
 nNum=nRemain;

 nResult=nNum/10000;
 nRemain=nNum-nResult*10000;
 SendChar(nResult+'0');
 nNum=nRemain;
```

```
 nResult=nNum/1000;
 nRemain=nNum-nResult*1000;
 SendChar(nResult+'0');
 nNum=nRemain;

 nResult=nNum/100;
 nRemain=nNum-nResult*100;
 SendChar(nResult+'0');
 nNum=nRemain;

 nResult=nNum/10;
 nRemain=nNum-nResult*10;
 SendChar(nResult+'0');
 SendChar(nRemain+'0');
}


#pragma DebugOff
void SendInt(int n){                    // prints a 2-byte integer on
int nNum, nRemain, nResult;             // the screen of host PC.

 if(n & 0x8000)
  {
   SendChar('-');
   nNum=(~n)+1;
   }
  else nNum=n;

 nResult=nNum/10000;
 nRemain=nNum-nResult*10000;
 SendChar(nResult+'0');
 nNum=nRemain;

 nResult=nNum/1000;
 nRemain=nNum-nResult*1000;
 SendChar(nResult+'0');
 nNum=nRemain;

 nResult=nNum/100;
 nRemain=nNum-nResult*100;
 SendChar(nResult+'0');
 nNum=nRemain;

 nResult=nNum/10;
 nRemain=nNum-nResult*10;
 SendChar(nResult+'0');
 SendChar(nRemain+'0');
// SendChar(10);
}
```

```
#pragma DebugOff
void SendIntBin(int n){
 SendChar(n>>8);
 SendChar(n & 0xFF);
}


#pragma DebugOff
void SendHexInt(int n){
int nNum;

 nNum=n>>12;
 if(nNum<10) SendChar(nNum + '0');
  else SendChar(nNum + 'A' - 10);

 nNum=n>>8;
 nNum &=0xF;
 if(nNum<10) SendChar(nNum + '0');
  else SendChar(nNum + 'A' - 10);

 nNum=n>>4;
 nNum &=0xF;
 if(nNum<10) SendChar(nNum + '0');
  else SendChar(nNum + 'A' - 10);

 nNum=n&0xF;
 if(nNum<10) SendChar(nNum + '0');
  else SendChar(nNum + 'A' - 10);
}


#pragma DebugOff
void SendChar(char Ch){
 S0TBUF=Ch;
 while(!S0TIR);
 S0TIC=0;
}


#pragma DebugOff
void SendStr(char *sz){
char ch;

 while(ch=*sz++) SendChar(ch);
}


#pragma DebugOff
char GetChar(void){

 S0RIR=0;
 S0RIC=0;
```

```
 return(S0RBUF);
}


#pragma DebugOff
char PeekChar(void){

 if(S0RIR) return(S0RBUF);
 return(0);
}
```

```
/* ------------ Reads166 generated header ------------
 * module description : T0init initializes Timer0,
 *                      T0Isr is the timer interrupt service
 *                      routine
 * module file name   : Timer01.c
 * ------------------------------------------------- */


// ---------------------------------------------------------------
// subroutine T0init initializes Timer0
// input    : timer 0 reload value
// output   : none
// ---------------------------------------------------------------

void T0init(int nReload){

 T0REL=nReload;          // reload value determines the period
 T0=nReload;             // start timer with reload value

// --- set up the timer control register ---
 T01CON=0;          //   prescalar=8, Timer0 off
 T0IC=0x45;         // enable T0 interrupts - priority level=1,
group=1

// --- initialize variables ---
 MSEC=0;                 // initialize ticks low word
 SECONDS=0;              // initialize seconds

// --- start timer ------------
 T0R=1;                  // start timer
}

// ---------------------------------------------------------------
------
// subroutine T0Isr Timer0 Interrupt Service Routine
//
// called upon Timer0 overflows
// ---------------------------------------------------------------
------
#pragma DebugOff
void interrupt(0x80) T0Isr(void){
 MSEC++;
 if(MSEC==100) {
   MSEC=0;
   SECONDS++;
 }
}
```

```
/* ------------ Reads166 generated header ------------
 * module description : Random Number Generator
 * module file name   : RandomParam.C
 * ---------------------------------------------------- */

unsigned int GetNewServiceTime(void) {
 float random;
 unsigned int uServiceTime;

 random=GetRandom();       // returns a random number 0-1
 uServiceTime=uServiceLow + random*(uServiceHigh-uServiceLow);
// SendStr("\ns");
// SendInt(uServiceTime);
 return uServiceTime;
}


unsigned int GetNewRepairTime(void) {
 float random;
 unsigned int uRepairTime;

 random=GetRandom();       // returns a random number 0-1
 uRepairTime=uRepairLow + random*(uRepairHigh-uRepairLow);
// SendStr("\nr");
// SendInt(uRepairTime);
 return uRepairTime;
}


unsigned int GetNewBreakDownTime(void) {
 float random;
 unsigned int uBreakDownTime;

 random=GetRandom();       // returns a random number 0-1
 uBreakDownTime=uBreakDownLow + random*(uBreakDownHigh-
uBreakDownLow);
// SendStr("\nb");
// SendInt(uBreakDownTime);
 return uBreakDownTime;
}


float GetRandom(void) {
 float rnd;

// SendInt(nSeed);
 nSeed=nSeed*2045+1;
 nSeed=nSeed-(nSeed/1048576)*1048576;
// SendLongInt(nSeed);
 rnd=(nSeed+1)/1045877.0;
 return rnd;
}
```

```
; ------------ Reads166 generated header ------------
; module description : CAN setup and message manipulation
;                      (assembly) routines
; module file name    : Can.asm
; ---------------------------------------------------

; --- constants ---
 REQUEST     equ  01h
 SHIPMENT    equ  02h
 PERMISSION  equ  03h


;-------------------------------------------------------------
can_init:

; --- set interrupt control register ---
   mov R0, #44h                  ; Interrupt request enabled
XP0IE=1,
   mov XP0IC, R0                 ; (Group Level) GL=0, (Int.
Priority Level) ILVL=1,

; --- start initialization ---
   calla cc_UC, start_init
   calla cc_UC, reset_all_obj

; ---  set bit timing register (500 kBaud) ---
   movb RH0, #15                 ; TSEG1=15
   movb RL0, #2                  ; TSEG2=2
   movb RH1, #0                  ; BRP=0
   movb RL1, #0                  ; SJW=0
   calla cc_UC, set_bit_timing

; --- set global mask registers ---
   mov  R0, #0FFFFh            ; upper mask word -> 1111 1111
1111 1111
   mov  UP_GL_MASK_LONG, r0    ; lower mask word -> 1111 1000
1111 1111
   mov  R0, #0F8FFh            ; all identifier bits of coming
messages
   mov  LO_GL_MASK_LONG, r0    ; are checked in this
configuration

; --- setup objects responsible for ID transfer ---
   calla cc_UC, setup_obj_7     ; Transmit obj (CAN NODE #)
   calla cc_UC, setup_obj_8     ; Rec obj (Largest CAN NODE #)
   calla cc_UC, setup_obj_9     ; Rec obj (Request for Operation
Stats)
   calla cc_UC, setup_obj_10    ; Transmit obj (Operation Stats)

; --- enable CAN interrupts ---
   calla cc_UC, set_ie
```

```
; --- finish initialization ---
   calla cc_UC, end_init

   ret



;-----------------------------------------------------------------
; objective : configures msg objects used for inter-nodal
;             communication.
; input     : none.
; output    : none.
; destroys  : nothing.
;-----------------------------------------------------------------
can_setup_messaging:
   calla cc_UC, start_init
   calla cc_UC, setup_obj_1      ; Receive obj (REQUEST)
   calla cc_UC, setup_obj_2      ; Receive obj (SHIPMENT)
   calla cc_UC, setup_obj_3      ; Receive obj (PERMISSION)
   calla cc_UC, setup_obj_4      ; Transmit obj (REQUEST)
   calla cc_UC, setup_obj_5      ; Transmit obj (SHIPMENT)
   calla cc_UC, setup_obj_6      ; Transmit obj (PERMISSION)
   calla cc_UC, end_init

   ret



;-----------------------------------------------------------------
setup_obj_1:
   push R0
   push R1
   push R2
   push R3

   cmp  R0, #1
   jmpr cc_ULE, done_1           ; if NodeID=1 then skip this
setup

   mov R3, R0                    ; save NodeID

; --- Object 1 (REQUEST receive) ---
   movb  RL0, #1                 ; obj #
; --- validate msg obj 1 ---
   calla cc_UC, set_msgval
; --- enable receive interrupt for msg obj 1 ---
   calla cc_UC, set_rxie
; --- set arbitration reg for msg obj 1 ---
   movb rH3, rL3                 ; source of Request is the
upstream
   subb rH3, #1                  ; node (i.e. NodeID-1)
```

```
   movb rL3, #REQUEST            ; ID for Request
   mov  R1, R3                   ; Arb. ID low word
   mov  R2, #0001h               ; Arb. ID high word
   calla cc_UC, set_arbitration
; --- configure msg obj 1 ---
   calla cc_UC, set_type_extd
   calla cc_UC, set_dir_receive
   movb rH0, #0
   calla cc_UC, set_dlc

done_1:
   pop R3
   pop R2
   pop R1
   pop R0
   ret


;---------------------------------------------------------------------
setup_obj_2:
   push R0
   push R1
   push R2
   push R3

   cmp R0, #1
   jmpr cc_ULE, done_2           ; if NodeID=1 then skip this
setup

   mov R3, R0                    ; save NodeID
; --- Object 2 (SHIPMENT receive) ---
   movb RL0, #2                  ; obj #
; --- validate msg obj 2 ---
   calla cc_UC, set_msgval
; --- enable receive interrupt for msg obj 2 ---
   calla cc_UC, set_rxie
; --- set arbitration reg for msg obj 2 ---
   movb rH3, rL3                 ; source of Shipment is the
upstream
   subb rH3, #1                  ; station (i.e. NodeID-1)
   movb rL3, #SHIPMENT           ; ID for Shipment
   mov  R1, R3                   ; Arb. ID low word
   mov  R2, #0001h               ; Arb. ID high word
   calla cc_UC, set_arbitration
; --- configure msg obj 2 ---
   calla cc_UC, set_type_extd
   calla cc_UC, set_dir_receive
   movb RH0, #0                  ; data length (bytes)
   calla cc_UC, set_dlc

done_2:
   pop R3
```

```
   pop R2
   pop R1
   pop R0
   ret


;-----------------------------------------------------------------
setup_obj_3:
   push R0
   push R1
   push R2
   push R3

   mov R3, R0                    ; save NodeID
; --- Object 3 (PERMISSION receive) ---
   movb RL0, #3                  ; obj #
; --- validate msg obj 3 ---
   calla cc_UC, set_msgval
; --- enable receive interrupt for msg obj 3 ---
   calla cc_UC, set_rxie
; --- set arbitration reg for msg obj 3 ---
   movb rH3, rL3                 ; source of Permission is the
downstream
   addb rH3, #1                  ; station (i.e. NodeID+1)
   movb rL3, #PERMISSION         ; ID for Permission
   mov  R1, R3                   ; Arb. ID low word
   mov  R2, #0001h               ; Arb. ID high word
   calla cc_UC, set_arbitration
; --- configure msg obj 3 ---
   calla cc_UC, set_type_extd
   calla cc_UC, set_dir_receive
   movb RH0, #0                  ; data length (bytes)
   calla cc_UC, set_dlc

   pop R3
   pop R2
   pop R1
   pop R0
   ret


;-----------------------------------------------------------------
setup_obj_4:
   push R0
   push R1
   push R2
   push R3

   mov R3, R0                    ; save NodeID
; --- Object 4 (REQUEST transmit) ---
   movb RL0, #4                  ; obj #
; --- validate msg obj 4 ---
```

```
   calla cc_UC, set_msgval
; --- set arbitration reg for msg obj 4 ---
   movb rH3, rL3                ; source of REQUEST is this
station
   movb rL3, #REQUEST           ; ID for Request
   mov  R1, R3                  ; Arb. ID low word
   mov  R2, #0001h              ; Arb. ID high word
   calla cc_UC, set_arbitration
; --- configure msg obj 4 ---
   calla cc_UC, set_type_extd
   calla cc_UC, set_dir_transmt
   movb RH0, #0                 ; data length (bytes)
   calla cc_UC, set_dlc

   pop R3
   pop R2
   pop R1
   pop R0
   ret


;---------------------------------------------------------------
setup_obj_5:
   push R0
   push R1
   push R2
   push R3

   mov R3, R0                   ; save NodeID
; --- Object 5 (SHIPMENT transmit) ---
   movb RL0, #5                 ; obj #
; --- validate msg obj 5 ---
   calla cc_UC, set_msgval
; --- set arbitration reg for msg obj 5 ---
   movb rH3, rL3                ; source of Shipment is this
station
   movb rL3, #SHIPMENT          ; ID for Shipment
   mov  R1, R3                  ; Arb. ID low word
   mov  R2, #0001h              ; Arb. ID high word
   calla cc_UC, set_arbitration
; --- configure msg obj 5 ---
   calla cc_UC, set_type_extd
   calla cc_UC, set_dir_transmt
   movb RH0, #0                 ; data length (bytes)
   calla cc_UC, set_dlc

   pop R3
   pop R2
   pop R1
   pop R0
   ret
```

```
;-----------------------------------------------------------------
setup_obj_6:
   push R0
   push R1
   push R2
   push R3

   cmp R0, #1
   jmpa cc_ULE, done_6              ; if NodeID=1 then skip this
setup

   mov R3, R0                       ; save NodeID
; --- Object 6 (PERMISSION transmit) ---
   movb RL0, #6                     ; obj #
; --- validate msg obj 6 ---
   calla cc_UC, set_msgval
; --- set arbitration reg for msg obj 6 ---
   movb rH3, rL3                    ; source of permission is this
object
   movb rL3, #PERMISSION            ; ID for Permission
   mov  R1, R3                      ; Arb. ID low word
   mov  R2, #0001h                  ; Arb. ID high word
   calla cc_UC, set_arbitration
; --- configure msg obj 6 ---
   calla cc_UC, set_type_extd
   calla cc_UC, set_dir_transmt
   movb RH0, #0                     ; data length (bytes)
   calla cc_UC, set_dlc

done_6:
   pop R3
   pop R2
   pop R1
   pop R0
   ret


;-----------------------------------------------------------------
setup_obj_7:
; --- Object 7 (CAN ID transmit) ---
   movb  RL0, #7                    ; obj #
; --- validate msg obj 7 ---
   calla cc_UC, set_msgval
; --- set arbitration reg for msg obj 7 ---
   mov  R1, #01h                    ; ID low word
   mov  R2, #00h                    ; ID high word
   calla cc_UC, set_arbitration
; --- configure msg obj 7 ---
   calla cc_UC, set_type_extd
   calla cc_UC, set_dir_transmt
   movb RH0, #1                     ; data length (bytes)
```

```
   calla cc_UC, set_dlc

   ret


;-------------------------------------------------------------
setup_obj_8:
; --- Object 8 (Last Node CAN ID receive) ---
   movb RL0, #8                ; obj #
; --- validate msg obj 8 ---
   calla cc_UC, set_msgval
   calla cc_UC, clr_cpuupd     ; clear MSGLST
; --- set arbitration reg for msg obj 8 ---
   mov  R1, #02h               ; ID low word
   mov  R2, #00h               ; ID high word
   calla cc_UC, set_arbitration
; --- configure msg obj 8 ---
   calla cc_UC, set_type_extd
   calla cc_UC, set_dir_receive
   movb RH0, #1                ; data length (bytes)
   calla cc_UC, set_dlc

   ret


;-------------------------------------------------------------
setup_obj_9:
; --- Object 9 (Request for Operation Stats, receive) ---
   movb RL0, #9                ; obj #
; --- validate msg obj 9 ---
   calla cc_UC, set_msgval
   calla cc_UC, clr_cpuupd     ; clear MSGLST
; --- set arbitration reg for msg obj 9 ---
   mov  R1, #01h               ; ID low word
   mov  R2, #02h               ; ID high word
   calla cc_UC, set_arbitration
; --- configure msg obj 9 ---
   calla cc_UC, set_type_extd
   calla cc_UC, set_dir_receive
   movb RH0, #1                ; data length (bytes)
   calla cc_UC, set_dlc

   ret


;-------------------------------------------------------------
setup_obj_10:
; --- Object 7 (Operation Stats transmit) ---
   movb  RL0, #10              ; obj #
; --- validate msg obj 10 ---
   calla cc_UC, set_msgval
; --- set arbitration reg for msg obj 10 ---
```

```
   mov  R1, #02h                ; ID low word
   mov  R2, #02h                ; ID high word
   calla cc_UC, set_arbitration
; --- configure msg obj 10 ---
   calla cc_UC, set_type_extd
   calla cc_UC, set_dir_transmt
   movb RH0, #7                 ; data length (bytes)
   calla cc_UC, set_dlc

   ret



;-------------------------------------------------------------
; objective : reads the Can ID in the first data byte.
; input     : object no in R0.
; returns   : CAN ID in R0.
; destroys  : R0.
;-------------------------------------------------------------
get_field_nodeid:
   push R1

   calla cc_UC, get_addr_data
   movb rH0, #0
   movb rL0, [R1]

   pop R1
   ret



;-------------------------------------------------------------
; objective : places the Can ID into the first data byte of
;             object 7.
; input     : Can ID in R0.
; returns   : nothing.
; destroys  : norhing.
;-------------------------------------------------------------
set_field_nodeid:
   push R0
   push R1
   push R2

   mov  R2, R0
   movb rL0, #7
   calla cc_UC, get_addr_data
   movb [R1], rL2

   pop R2
   pop R1
   pop R0
   ret
```

```
;-----------------------------------------------------------------
; objective : places the production count in this interval into
;             the data bytes 1-2 (H-L) of object 10.
; input     : production count (output) in R0.
; returns   : nothing.
; destroys  : nothing.
;-----------------------------------------------------------------
set_field_prodcount:
   push R0
   push R1
   push R2

   mov  R2, R0
   movb rL0, #10
   calla cc_UC, get_addr_data
   movb [R1], rH2
   add R1, #1
   movb [R1], rL2

   pop R2
   pop R1
   pop R0
   ret



;-----------------------------------------------------------------
; objective : places inventory cost in this interval into data
;             bytes 3-6 (H-L) of object 10.
; input     : inventory cost in R1 and R0 (H-L).
; returns   : nothing.
; destroys  : nothing.
;-----------------------------------------------------------------
set_field_invcost:
   push R0
   push R1
   push R2
   push R3

   mov  R3, R1
   mov  R2, R0
   movb rL0, #10
   calla cc_UC, get_addr_data
   add R1, #2
   movb [R1], rH3
   add R1, #1
   movb [R1], rL3
   add R1, #1
   movb [R1], rH2
   add R1, #1
```

```
   movb [R1], rL2

   pop R3
   pop R2
   pop R1
   pop R0
   ret



;-----------------------------------------------------------------
; objective : places the buffersize in this interval into data
byte 7
;             of object 10.
; input     : buffersize in R0.
; returns   : nothing.
; destroys  : norhing.
;-----------------------------------------------------------------
set_field_buffersize:
   push R0
   push R1
   push R2

   mov  R2, R0
   movb rL0, #10
   calla cc_UC, get_addr_data
   add R1, #6
   movb [R1], rL2

   pop R2
   pop R1
   pop R0
   ret
```

Source Code for Scanner Module

The program running in the scanner module consists of the following files:

- Code.c - contains the main program.
- CanMessage.c - contains code for CAN message transmission.
- Canisr.c - contains an interrupt service routine.
- S0.c - contains the string output routines.
- Timer01.c - implements timer interrupt for time base.
- Can.asm - contains assembly code for CAN setup

Of the above, S0.c and Timer01.c are the same as the ones running in station controllers. The others contain code specific to the operation of the scanner.

Code.c is the main program module. This is where the initialization of the scanner and CAN setup is done. For CAN setup, *Code.c* calls *Can.asm*. After the initialization and setup phase, the program in *Code.c* basically counts clock ticks until the end of an interval. At the end of each interval, it polls (he stations (ReceiveStats routine) in order to get their production data. It records the transmitted data (Evaluation routine) and also prints them on the screen (PrintStats routine). The reception of production data from the stations in the network is implemented in an interrupt driven fashion. *Canisr.c* module contains this code.

```
/* ------------ Reads166 generated header ------------
 * module description : main program
 * module file name   : Code.C
 * ---------------------------------------------------- */

// --- Constants ---
#define INTERVALLENGTH 100
#define PRODUCTREWARD 1000


// --- Variables ---
predef unsigned int T0, T0REL, T01CON, T0IC, S0RIC;
predef unsigned int INTREG, CTLST;
predef unsigned int  MSGCTL1, MSGCTL2, MSGCTL3, MSGCTL7;
predef unsigned bit T0R, S0RIR, IEN, DP2_0, P2_0;

unsigned int MSEC, SECONDS;
unsigned int uSeconds;
unsigned int uMessageType;
unsigned int uCanNodeID, uLastNodeID=0;
unsigned int uBufferSize[20];
unsigned int uIntervalNo;

long nProfit[20], nProdCount[20], nInvCost[20];
long nTotalProdCount[20];
long nLineProfit, nLineRevenue, nLineInvCost;
long nTotalLineProfit, nTotalLineRevenue, nTotalLineInvCost;

// --- Enumerations ---
enum uMessageType {IDREQUEST=1,STATREQUEST,LASTNODEID};

// --- C Functions ---
void InitializeVars(void);
void StartEmulation(void);
void PauseEmulation(void);
void ReceiveStats(void);
void PrintStats(void);
void PrintFinalStats(void);
void Evaluate(void);
void SendMessage(unsigned int);
void Delay_sec(unsigned int);

// --- Assembly Functions ---
unsigned int get_field_nodeid(unsigned int);
unsigned int get_field_prodcount(unsigned int);
long get_field_invcost(unsigned int);
unsigned int get_field_buffersize(unsigned int);
void wait_msg(unsigned int);
void clr_newdat(unsigned int);
void set_field_nodeid(unsigned int,unsigned int);
void can_init(void);
```

```
main(void){

 SendStr("\nPress Enter to start !");
 while(!PeekChar());           // wait for a char

 S0RIC=0;

 SendStr("\n\nNetwork Initialization started ...");
 can_init();                   // initialize CAN
 IEN=1;                        // enable interrupts
 SendStr("\n\n CAN Setup completed!");
 Delay_sec(1);
 SendMessage(IDREQUEST);    // ask nodes to report their CAN ID's
 SendStr("\n Handshake Sequence started!");

 Delay_sec(3);
 SendStr("\n Handshake Sequence completed!");
 SendStr("\n    *** CAN Network has ");
 SendInt(uLastNodeID);
 SendStr(" nodes ***");

 Delay_sec(1);
 set_field_nodeid(2,uLastNodeID);
 SendMessage(LASTNODEID);
 SendStr("\n Broadcasting the Last_Node_ID ");
 SendInt(uLastNodeID);

 Delay_sec(1);
 InitializeVars();
 SendStr("\n\nNetwork Initialization completed ...");
 DP2_0=1;                      // configure Port 2_0 as output
 P2_0=0;
 Delay_sec(1);

 uIntervalNo=0;
 SendStr("\n\nProduction starting ...\n");
 do {
   uSeconds=0;
   uIntervalNo++;
   StartEmulation();
   do {
     if (uSeconds!=SECONDS) {
       uSeconds=SECONDS;
       if (!(uSeconds%10)) {
         SendChar('.');
       }
     }
   } while (uSeconds<INTERVALLENGTH);
   PauseEmulation();
   Delay_sec(1);
   ReceiveStats();
```

```
   Evaluate();
   PrintStats();
 } while(uIntervalNo<100);
 PrintFinalStats();
}


void ReceiveStats(void) {
unsigned int i;

 for (i=0; i<uLastNodeID; i++) {
   SendStr("\n Node ");
   SendInt(i);
   set_field_nodeid(3,i+1);
   SendMessage(STATREQUEST);
   SendStr(" request sent...");
   wait_msg(4);
   SendStr(" stats received...");
   nProdCount[i]=get_field_prodcount(4);
   nInvCost[i]=get_field_invcost(4);
   uBufferSize[i]=get_field_buffersize(4);
   clr_newdat(4);
   Delay_sec(1);
 }
}


void StartEmulation(void) {
  SendStr("\n\n Interval ");
  SendInt(uIntervalNo);
  P2_0=1;                  // send START signal
  T0init(0x0A000);          // start emulation clock, initialize
Timer0
}


void PauseEmulation(void) {
  T0IC=0;                  // stop emulation clock
  P2_0=0;                  // send STOP signal
}


void Evaluate(void) {
int i;

 nLineRevenue=nProdCount[uLastNodeID-1]*PRODUCTREWARD;
 nLineInvCost=0;
 for (i=0; i<uLastNodeID; i++) {
   nLineInvCost+=nInvCost[i];
   nTotalProdCount[i]+=nProdCount[i];
 }
 nLineProfit=nLineRevenue-nLineInvCost;
```

```
 nTotalLineInvCost+=nLineInvCost;
 nTotalLineRevenue+=nLineRevenue;
 nTotalLineProfit+=nLineProfit;
}


void PrintStats(void) {
int i;

 SendStr("\n                Count     Inv Cost  Buffer");
 SendStr("\n              ---------- ---------- ------");
 for (i=1; i<=uLastNodeID; i++) {
   SendStr("\nNode ");
   SendInt(i);
   SendStr(" : ");
   SendLongInt(nProdCount[i-1]);
   SendChar(' ');
   SendLongInt(nInvCost[i-1]);
   SendChar(' ');
   SendInt(uBufferSize[i-1]);
 }
}


void PrintFinalStats(void) {
  SendStr("\n\n  Final Production Stats");
  SendStr("\n ------------------------");
  SendStr("\n Parts Produced : ");
  SendLongInt(nTotalProdCount[uLastNodeID-1]);
  SendStr("\n Revenue        : ");
  SendLongInt(nTotalLineRevenue);
  SendStr("\n Inventory Cost : ");
  SendLongInt(nTotalLineInvCost);
  SendStr("\n Profit         : ");
  SendLongInt(nTotalLineProfit);
}


void Delay_sec(unsigned int uSec) {
 T0init(0x6000);                    // initialize Timer0
 while(SECONDS<uSec);               // wait until all the nodes
respond
 T0IC=0;
}


void InitializeVars(void) {
 int i;

 nTotalLineProfit=0;
 nTotalLineRevenue=0;
 nTotalLineInvCost=0;
```

```
 for (i=0; i<uLastNodeID; i++) {
   nTotalProdCount[i]=0;
 }
}
```

```
/* ------------ Reads166 generated header ------------
 * module description : routines for CAN message transmission
 * module file name   : CanMessage.c
 * ---------------------------------------------------- */


void out_msg(unsigned int);
void out_rmt_rqst(unsigned int);


void SendMessage(unsigned int uMessageType) {

  if (uMessageType==IDREQUEST) {
    out_rmt_rqst(1);
  }
  else if (uMessageType==LASTNODEID) {
    TransmitObj(2);
  }
  else if (uMessageType==STATREQUEST) {
    TransmitObj(3);
  }
}


void TransmitObj(unsigned int nObjNo) {
int nTransmissionCount=0;
  do {
    out_msg(nObjNo);      // call assembly routine in Rcanlib.inc
    nTransmissionCount++;
  } while(TransmissionError() && nTransmissionCount<3);
}


unsigned int TransmissionError(void) {
unsigned int uErrorStatus;
  uErrorStatus=CTLST&&0x0700;
  if (uErrorStatus==3) {
    SendStr(" NAck");
  }
  return(uErrorStatus);
}
```

```
/* ------------ Reads166 generated header ------------
 * module description : interrupt service routine for the
 *                      reception of Transmit_ID message
 * module file name   : Canisr.c
 * ------------------------------------------------------ */

predef unsigned int INTREG;

void interrupt(0x100) CANIsr(void){
unsigned int nIntID;

  nIntID=INTREG&0x00FF;      // read CAN Interrupt Register to
                             // find out the interrupt source
  if (nIntID==3) {           // Can ID received
    uCanNodeID=get_field_nodeid(1);
    if(uCanNodeID>uLastNodeID)  uLastNodeID=uCanNodeID;
    MSGCTL1=0xFCFF;          // clear NEWDAT
    MSGCTL1=0xFFFD;          // clear INTPND
    SendStr("\n  Link established with Node ");
    SendInt(uCanNodeID);
  }
}
```

```
; ------------ Reads166 generated header ------------
; module description : CAN setup and message manipulation
;                      (assembly) routines
; module file name    : Can.asm
; ---------------------------------------------------

can_init:

; --- set interrupt control register ---
   mov R0, #44h                  ; Interrupt request enabled
XP0IE=1,
   mov XP0IC, R0                 ; (Group Level) GL=0, (Int.
Priority Level) ILVL=1,

; --- start initialization ---
   calla cc_UC, start_init
   calla cc_UC, reset_all_obj

; ---  set bit timing register (500 kBaud) ---
   movb RH0, #15                 ; TSEG1=15
   movb RL0, #2                  ; TSEG2=2
   movb RH1, #0                  ; BRP=0
   movb RL1, #0                  ; SJW=0
   calla cc_UC, set_bit_timing

; --- set global mask registers ---
   mov  R0, #0FFFFh              ; upper mask word -> 1111 1111
                                 ;                    1111 1111
   mov  UP_GL_MASK_LONG, r0      ; lower mask word -> 1111 1000
                                 ;                    1111 1111
   mov  R0, #0F8FFh              ; all identifier bits of the
                                 ; incoming messages are checked
   mov  LO_GL_MASK_LONG, r0      ; in this configuration

   calla cc_UC, setup_obj_1      ; Receive obj (Remote Request)
   calla cc_UC, setup_obj_2      ; Transmit obj (Last Can ID)
   calla cc_UC, setup_obj_3      ; Transmit obj (Request for
                                 ; Operation Stats)
   calla cc_UC, setup_obj_4      ; Receive obj (Operation Stats)

; --- enable CAN interrupts ---
   calla cc_UC, set_ie

; --- finish initialization ---
   calla cc_UC, end_init

   ret
```

```
; ------------------------------------------------
setup_obj_1:
; --- Object 1 (Receive - Remote Request) ---
   movb  RL0, #1                ; obj #
; --- validate msg obj 1 ---
   calla cc_UC, set_msgval
; --- enable receive interrupt for msg obj 1 ---
   calla cc_UC, set_rxie
; --- set arbitration reg for msg obj 1 ---
   mov  R1, #00h                ; ID low word
   mov  R2, #01h                ; ID high word
   calla cc_UC, set_arbitration
; --- configure msg obj 1 ---
   calla cc_UC, set_type_extd
   calla cc_UC, set_dir_receive
   movb rH0, #1
   calla cc_UC, set_dlc

   ret


; ------------------------------------------------
setup_obj_2:
; --- Object 2 (Last Can ID, transmit) ---
   movb RL0, #2                 ; obj #
; --- validate msg obj 2 ---
   calla cc_UC, set_msgval
; --- set arbitration reg for msg obj 2 ---
   mov  R1, #00h                ; ID low word
   mov  R2, #02h                ; ID high word
   calla cc_UC, set_arbitration
; --- configure msg obj 2 ---
   calla cc_UC, set_type_extd
   calla cc_UC, set_dir_transmt
   movb RH0, #1                 ; data length (bytes)
   calla cc_UC, set_dlc

   ret


; ------------------------------------------------
setup_obj_3:
; --- Object 3 (Request for Stats, transmit) ---
   movb RL0, #3                 ; obj #
; --- validate msg obj 3 ---
   calla cc_UC, set_msgval
; --- set arbitration reg for msg obj 3 ---
   mov  R1, #00h                ; ID low word
   mov  R2, #03h                ; ID high word
   calla cc_UC, set_arbitration
; --- configure msg obj 3 ---
   calla cc_UC, set_type_extd
```

```
   calla cc_UC, set_dir_transmt
   movb RH0, #1                    ; data length (bytes)
   calla cc_UC, set_dlc

   ret


; ------------------------------------------------
setup_obj_4:
; --- Object 4 (Operation Stats, receive) ---
   movb  RL0, #4                   ; obj #
; --- validate msg obj 4 ---
   calla cc_UC, set_msgval
; --- set arbitration reg for msg obj 4 ---
   mov  R1, #00h                   ; ID low word
   mov  R2, #04h                   ; ID high word
   calla cc_UC, set_arbitration
; --- configure msg obj 4 ---
   calla cc_UC, set_type_extd
   calla cc_UC, set_dir_receive
   movb rH0, #7
   calla cc_UC, set_dlc

   ret


;-----------------------------------------------------------------
; objective : reads the Can ID in the first data byte.
; input     : object no in R0.
; returns   : CAN ID in R0.
; destroys  : R0.
;-----------------------------------------------------------------
get_field_nodeid:
   push R1

   calla cc_UC, get_addr_data
   movb rH0, #0
   movb rL0, [R1]

   pop R1
   ret


;-----------------------------------------------------------------
; objective : reads the production count in data bytes 1-2 (H-L).
; input     : object no in R0.
; returns   : production count in R0.
; destroys  : R0.
;-----------------------------------------------------------------
get_field_prodcount:
   push R1
```

```
   calla cc_UC, get_addr_data
   movb rH0, [R1+]
   movb rL0, [R1]

   pop R1
   ret


;-------------------------------------------------------------
; objective : reads the inventory cost in data bytes 3-6 (H-L).
; input     : object no in R0.
; returns   : profit in R1 and R0 (H-L).
; destroys  : R1 and R0.
;-------------------------------------------------------------
get_field_invcost:
   push R2

   calla cc_UC, get_addr_data
   add R1, #2

   movb rH2, [R1+]
   movb rL2, [R1+]

   movb rH0, [R1+]
   movb rL0, [R1]

   mov R1, R2

   pop R2
   ret


;-------------------------------------------------------------
; objective : reads the buffersize in data byte 7.
; input     : object no in R0.
; returns   : buffersize in R0.
; destroys  : R0.
;-------------------------------------------------------------
get_field_buffersize:
   push R1

   calla cc_UC, get_addr_data
   add R1, #6
   movb rL0, [R1]
   movb rH0, #0

   pop R1
   ret
```

```
;-----------------------------------------------------------------
; objective : places the Can ID of the last node into the first
;             data byte of object in R0.
; input     : object no in R0, Can ID in R1.
; returns   : nothing.
; destroys  : norhing.
;-----------------------------------------------------------------
set_field_nodeid:
   push R0
   push R1
   push R2

   mov  R2, R1
   calla cc_UC, get_addr_data
   movb [R1], rL2

   pop R2
   pop R1
   pop R0
   ret
```

```
; ------------ Reads166 generated header ------------
; module description :
; module file name   : Rcanlib.inc
; ---------------------------------------------------

;-------------------------------------------------------------------
; constant declarations
;-------------------------------------------------------------------

; CAN interrupt vector
  XP0INT          equ   100h

; control register for XP0INT
  XP0IC           equ  0F186h

; control/status register
  CTL_ST          equ 0EF00h

; interrupt register
  INT_REG         equ 0EF02h

; bit timing register
  BIT_TIMING      equ 0EF04h

; global mask short
  GL_MASK_SH      equ 0EF06h

; upper global mask long
  UP_GL_MASK_LONG equ 0EF08h

; lower global mask long
  LO_GL_MASK_LONG equ 0EF0Ah

; message control registers
  MSG_CTL_1       equ 0EF10h
  MSG_CTL_2       equ 0EF20h
  MSG_CTL_3       equ 0EF30h
  MSG_CTL_4       equ 0EF40h
  MSG_CTL_5       equ 0EF50h
  MSG_CTL_6       equ 0EF60h
  MSG_CTL_7       equ 0EF70h
  MSG_CTL_8       equ 0EF80h
  MSG_CTL_9       equ 0EF90h
  MSG_CTL_10      equ 0EFA0h
  MSG_CTL_11      equ 0EFB0h
  MSG_CTL_12      equ 0EFC0h
  MSG_CTL_13      equ 0EFD0h
```

```
  MSG_CTL_14       equ 0EFE0h
  MSG_CTL_15       equ 0EFF0h

; offsets
  UP_ARBITR_OFF    equ 2
  LO_ARBITR_OFF    equ 4
  MSG_CONFG_OFF    equ 6
  DATA_OFF         equ 7
;-------------------------------------------------------------------
;-------------------------------------------------------------------
;
; List of Routines:
;
; board level
;---------------
; start_init
; reset_all_obj
; set_bit_timing
; set_mask_short
; set_mask_long      ?????
; end_init
; can_err_status
; set_ie
; clr_txok
; clr_rxok
; clr_intr_rqst
; clr_intr_reg
;
; object level
;---------------
; reset_obj
; clr_rmtpnd
; set_txrq
; clr_txrq
; set_cpuupd
; clr_cpuupd
; set_newdat
; clr_newdat
; set_msgval
; clr msgval
; set_txie
; clr_txie
; set_rxie
; clr_rxie
; set_intpnd
; clr_intpnd
; set_arbitration
; set_dlc
; get_dlc
; set_dir_transmt
; set_dir_receive
; set_type_extd
```

```
; set_type_std
;
; message level
;---------------
; in_rmt_rqst
; wait_rmt_rqst
; in_msg
; wait_msg
; out_rmt_rqst
; out_msg
; copydat_mem_reg
; copydat_reg_mem
;
; address
;---------------
; get_addr_ctl
; get_addr_up_arb
; get_addr_lo_arb
; get_addr_confg
; get_addr_data




;-----------------------------------------------------------------
; Function: starts the initialization of the CAN controller.
; Input:    none.
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
start_init:
   push R0

   mov R0, CTL_ST
   or R0, #0001h           ; set the INIT bit of the
control/status
   mov CTL_ST, R0          ; register.

   pop R0
   ret


;-----------------------------------------------------------------
; Function: resets objects 1-14.
; Input:    none.
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
reset_all_obj:
   push R0

   movb rl0, #1            ; declare all message objects invalid.
```

```
   calla cc_UC, reset_obj
   movb rl0, #2
   calla cc_UC, reset_obj
   movb rl0, #3
   calla cc_UC, reset_obj
   movb rl0, #4
   calla cc_UC, reset_obj
   movb rl0, #5
   calla cc_UC, reset_obj
   movb rl0, #6
   calla cc_UC, reset_obj
   movb rl0, #7
   calla cc_UC, reset_obj
   movb rl0, #8
   calla cc_UC, reset_obj
   movb rl0, #9
   calla cc_UC, reset_obj
   movb rl0, #10
   calla cc_UC, reset_obj
   movb rl0, #11
   calla cc_UC, reset_obj
   movb rl0, #12
   calla cc_UC, reset_obj
   movb rl0, #13
   calla cc_UC, reset_obj
   movb rl0, #14
   calla cc_UC, reset_obj

   pop R0
   ret


;-----------------------------------------------------------------
; Function: configures the bit timing register.
; Inputs:   TSEG2 value in rl0
;           TSEG1 value in rh0
;           SJW   value in rl1
;           BRP   value in rh1
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
set_bit_timing:

   push r2

   mov r2, #0040h            ; set the CCE bit of the
control/status
   or CTL_ST, r2             ; register to gain access to the bit
                             ; timing register.
   mov r2, #0
   addb rl2, rl0             ; set TSEG2 field
```

```
   shl r2, #4
   addb r12, rh0              ; set TSEG1 field
   shl r2, #2
   addb r12, rl1              ; set SJW field
   shl r2, #6
   addb r2, rh1               ; set BRP field
   mov BIT_TIMING, r2
   mov r2, #0FFBFh            ; clear the CCE bit of the
control/status
   and CTL_ST, r2             ; register to inhibit further access
to
                              ; the bit timing register
   pop r2
   ret



;---------------------------------------------------------------
; Function: configures the global mask for messages with standard
;           identifier. The short (standard) mask consists of 11
;           bits.
; Inputs:   bits 0-7 of the mask in rl0.   (MMMM MMMM)
;           bits 8-10 of the mask in rh0   (XXXX XMMM)
;                              M=mask bit, X=don't care
; Output:   none.
; Destroys: none.
;---------------------------------------------------------------
set_mask_short:

   push r0
   or r0, #0F800h
   ror r0, #3
   mov GL_MASK_SH, r0
   pop r0
   ret



;---------------------------------------------------------------
; Function: finishes the initialization of the CAN controller.
; Input:    none.
; Output:   none.
; Destroys: none.
;---------------------------------------------------------------
end_init:

   push r0
   mov r0, #0FFFEh      ; clear the INIT bit of the control/status
   and CTL_ST, r0       ; register to complete the initialization.
   pop r0
   ret
```

```
;-----------------------------------------------------------------
; Function: displays the error status of the CAN unit. If the CAN
;           unit is off the bus (i.e. BOFF bit of control/status
;           register is set), or if one of the error counters
;           have reached the error warning limit (i.e. EWRN bit
;           is set), then this routine prints an error message.
; Input:    none.
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
can_err_status:

   push R1

   mov R1, CTL_ST
   andb RH1, #80h
   cmpb RH1, #80h
   jmpr cc_NE, is_ewrn
   mov R1, #bus_off_msg
   calla cc_UC, print
   pop R1
   ret
is_ewrn:
   mov R1, CTL_ST
   andb RH1, #40h
   cmpb RH1, #40h
   jmpr cc_NE, no_err
   mov R1, #ewrn_msg
   calla cc_UC, print
no_err:
   pop R1
   ret


bus_off_msg:
   db "BUS OFF !   Reset board", 0dh, 0ah, 0

ewrn_msg:
   db "ERROR WARNING !   Reset board", 0dh, 0ah, 0


;-----------------------------------------------------------------
; Function: clears the interrupt register.
; Input:    none.
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
clr_intr_reg:
   push R0

   movb RL0, #00h
   movb INT_REG, RL0
```

```
   pop R0
   ret

;-------------------------------------------------------------------
; Function: sets the IE (interrupt enable) bit of the
;           control/status register.
; Input:    none.
; Output:   none.
; Destroys: none.
;-------------------------------------------------------------------
set_ie:
   push R0

   mov R0, CTL_ST
   or R0, #0002h
   mov CTL_ST, R0

   pop R0
   ret


;-------------------------------------------------------------------
; Function: clears the TXOK bit of the control/status register.
; Input:    none.
; Output:   none.
; Destroys: none.
;-------------------------------------------------------------------
clr_txok:
   push r0

   mov r0, CTL_ST
   and r0, #0F7FFh
   mov CTL_ST, r0

   pop r0
   ret



;-------------------------------------------------------------------
; Function: clears the RXOK bit of the control/status register.
; Input:    none.
; Output:   none.
; Destroys: none.
;-------------------------------------------------------------------
clr_rxok:
   push r0

   mov r0, CTL_ST
   and r0, #0EFFFh
```

```
   mov CTL_ST, r0

   pop r0
   ret


;---------------------------------------------------------------
; Function: clears the Interrupt Request bit of the Interrupt
;           Control Register (XP0IC).
; Input:    none.
; Output:   none.
; Destroys: none.
;---------------------------------------------------------------
clr_intr_rqst:

   push R0

   mov R0, XP0IC
   andb RL0, #7Fh
   mov XP0IC, R0

   pop R0
   ret


;---------------------------------------------------------------
; Function: clears (RMTPND, TXRQ, NEWDAT, MSGVAL, TXIE, RXIE,
;           INTPND) fields and sets CPUUPD field of the control
;           register of the message object.
; Input:    object number in rl0.
; Output:   none.
; Destroys: none.
;---------------------------------------------------------------
reset_obj:

   calla cc_UC, clr_rmtpnd
   calla cc_UC, clr_txrq
   calla cc_UC, set_cpuupd
   calla cc_UC, clr_newdat
   calla cc_UC, clr_msgval
   calla cc_UC, clr_txie
   calla cc_UC, clr_rxie
   calla cc_UC, clr_intpnd
   ret
```

```
;-----------------------------------------------------------------
; Function: clears the RMTPND field of the message control
;           register.
; Input:    object number in r10.
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
clr_rmtpnd:

   push r0
   push r1
   calla cc_UC, get_addr_ctl
   mov r0, #7FFFh
   mov [r1], r0
   pop r1
   pop r0
   ret


;-----------------------------------------------------------------
; Function: sets the TXRQ field of the message control register.
; Input:    object number in R0.
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
set_txrq:

   push R0
   push R1
   calla cc_UC, get_addr_ctl
   mov R0, #0EFFFh
   mov [R1], R0
   pop R1
   pop R0
   ret


;-----------------------------------------------------------------
; Function: clears the TXRQ field of the message control
;           register.
; Input:    object number in R0.
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
clr_txrq:

   push R0
   push R1
   calla cc_UC, get_addr_ctl
   mov R0, #0DFFFh
   mov [R1], R0
   pop R1
```

```
   pop R0
   ret


;-------------------------------------------------------------
; Function: sets the CPUUPD field of the message control
;           register.
; Input:    object number in R0.
; Output:   none.
; Destroys: none.
;-------------------------------------------------------------
set_cpuupd:

   push R0
   push R1
   calla cc_UC, get_addr_ctl
   mov R0, #0FBFFh
   mov [R1], R0
   pop R1
   pop R0
   ret


;-------------------------------------------------------------
; Function: clears the CPUUPD field of the message control
;           register.
; Input:    object number in R0.
; Output:   none.
; Destroys: none.
;-------------------------------------------------------------
clr_cpuupd:

   push R0
   push R1
   calla cc_UC, get_addr_ctl
   mov R0, #0F7FFh
   mov [R1], R0
   pop R1
   pop R0
   ret


;-------------------------------------------------------------
; Function: sets the NEWDAT field of the message control
;           register.
; Input:    object number in R0.
; Output:   none.
; Destroys: none.
;-------------------------------------------------------------
set_newdat:

   push R0
```

```
   push R1
   calla cc_UC, get_addr_ctl
   mov R0, #0FEFFh
   mov [R1], R0
   pop R1
   pop R0
   ret


;---------------------------------------------------------------
; Function: clears the NEWDAT field of the message control
;           register.
; Input:    object number in R0.
; Output:   none.
; Destroys: none.
;---------------------------------------------------------------
clr_newdat:

   push R0
   push R1
   calla cc_UC, get_addr_ctl
   mov R0, #0FDFFh
   mov [R1], R0
   pop R1
   pop R0
   ret



;---------------------------------------------------------------
; Function: declares the message object valid by setting the
;           MSGVAL field of the message control register.
; Input:    object number in R0.
; Output:   none.
; Destroys: none.
;---------------------------------------------------------------
set_msgval:

   push R0
   push R1
   calla cc_UC, get_addr_ctl
   mov R0, #0FFBFh
   mov [R1], R0
   pop R1
   pop R0
   ret
```

```
;-----------------------------------------------------------------
; Function: declares the message object invalid by clearing the
;           MSGVAL field of the message control register.
; Input:    object number in R0.
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
clr_msgval:

   push R0
   push R1
   calla cc_UC, get_addr_ctl
   mov R0, #0FF7Fh
   mov [R1], R0
   pop R1
   pop R0
   ret


;-----------------------------------------------------------------
; Function: sets the TXIE field of the message control register.
; Input:    object number in rl0.
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
set_txie:

   push r0
   push r1
   calla cc_UC, get_addr_ctl
   mov r0, #0FFEFh
   mov [r1], r0
   pop r1
   pop r0
   ret


;-----------------------------------------------------------------
; Function: clears the TXIE field of the message control
;           register.
; Input:    object number in rl0.
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
clr_txie:

   push r0
   push r1
   calla cc_UC, get_addr_ctl
   mov r0, #0FFDFh
   mov [r1], r0
   pop r1
```

```
   pop r0
   ret


;---------------------------------------------------------------
; Function: sets the RXIE field of the message control register.
; Input:    object number in rl0.
; Output:   none.
; Destroys: none.
;---------------------------------------------------------------
set_rxie:

   push r0
   push r1
   calla cc_UC, get_addr_ctl
   mov r0, #0FFFBh
   mov [r1], r0
   pop r1
   pop r0
   ret


;---------------------------------------------------------------
; Function: clears the RXIE field of the message control
;           register.
; Input:    object number in rl0.
; Output:   none.
; Destroys: none.
;---------------------------------------------------------------
clr_rxie:

   push r0
   push r1
   calla cc_UC, get_addr_ctl
   mov r0, #0FFF7h
   mov [r1], r0
   pop r1
   pop r0
   ret


;---------------------------------------------------------------
; Function: sets the INTPND field of the message control
;           register.
; Input:    object number in rl0.
; Output:   none.
; Destroys: none.
;---------------------------------------------------------------
set_intpnd:

   push r0
   push r1
```

```
   calla cc_UC, get_addr_ctl
   mov r0, #0FFFEh
   mov [r1], r0
   pop r1
   pop r0
   ret


;---------------------------------------------------------------
; Function: clears the INTPND field of the message control
;           register.
; Input:    object number in rl0.
; Output:   none.
; Destroys: none.
;---------------------------------------------------------------
clr_intpnd:

   push r0
   push r1
   calla cc_UC, get_addr_ctl
   mov r0, #0FFFDh
   mov [r1], r0
   pop r1
   pop r0
   ret


;---------------------------------------------------------------
; Function: configures the arbitration registers of the message
;           object.
; Inputs:   object number in rl0 (range: 1-15)
;           lower arbitration word in r1 (arbitration bits 0-15)
;           upper arbitration word in r2 (arbitration bits 16-28)
;           r1 -> AAAA AAAA AAAA AAAA ; r2 -> XXXA AAAA AAAA AAAA
;                 A=arbitration bit ; X=don't care
; Output:   none.
; Destroys: none.
;---------------------------------------------------------------
set_arbitration:

   push r1
   push r2
   push r3

   and r2, #1FFFh              ; clear three MSB in r2
   mov r3, r1
   and r3, #0E000h             ; isolate arbitration bits 13-15
   or r2, r3
   ror r2, #5                  ; r2 has the upper arbitration word

   and r1, #1FFFh              ; clear three MSB in r1
   ror r1, #5                  ; rotate right by 5
```

```
   mov r3, r1                  ; r3 has the lower arbitration word

   calla cc_UC, get_addr_up_arb
   mov [r1], r2
   calla cc_UC, get_addr_lo_arb
   mov [r1], r3

   pop r3
   pop r2
   pop r1
   ret


;-----------------------------------------------------------------
; Function: sets the DLC (Data Length Code) field in the
;           configuration register of the message object.
; Inputs:   object number is passed in rl0. Its value must be
;           between 1 and 15.
;           the DLC value is passed in rh0. The valid range for
;           DLC is 0 to 8.
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
set_dlc:

   push r1
   push r2
   push r3

   movb rl3, rh0                ; copy DLC input into rl3
   shl r3, #4                   ; adjust DLC input
   calla cc_UC, get_addr_confg  ; get address of the confg. reg.
   movb rl2, [r1]               ; copy confg. reg. into rl2
   andb rl2, #0Ch            ; clear bits 4-7 (DLC field) and 0-1
   addb rl2, rl3             ; set the DLC field of the confg.
   movb [r1], rl2            ; register

   pop r3
   pop r2
   pop r1
   ret



;-----------------------------------------------------------------
; Function: returns the DLC value in rl1.
;-----------------------------------------------------------------
get_dlc:

   push r2
   calla cc_UC, get_addr_confg
   movb rl2, [r1]
```

```
   and r2, #00F0h
   shr r2, #4
   mov r1, r2
   pop r2
   ret


;-----------------------------------------------------------------
; Function: sets the DIR (direction) bit in the configuration
;           register of the message object.
; Inputs:   object number is passed in rl0. Its value must be
;           between 1 and 15.
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
set_dir_transmt:

   push r0
   push r1

   calla cc_UC, get_addr_confg   ; get address of the confg. reg.
   movb rl0, [r1]                 ; copy confg. reg. into rl0
   orb rl0, #08h                 ; sets the direction bit (bit 3)
   movb [r1], rl0                 ; of the configuration register

   pop r1
   pop r0
   ret


;-----------------------------------------------------------------
; Function: clears the DIR (direction) bit in the configuration
;           register of the message object.
; Inputs:   object number is passed in rl0. Its value must be
;           between 1 and 15.
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
set_dir_receive:

   push r0
   push r1

   calla cc_UC, get_addr_confg   ; get address of the confg. reg.
   movb rl0, [r1]              ; copy confg. reg. into rl0
   andb rl0, #0F4h             ; clears the direction bit (bit 3)
   movb [r1], rl0              ; of the configuration register

   pop r1
   pop r0
   ret
```

```
;-----------------------------------------------------------------
; Function: sets the XTD (type) bit in the configuration register
;           of the message object.
; Inputs:   object number is passed in rl0. Its value must be
;           between 1 and 15.
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
set_type_extd:

   push r0
   push r1

   calla cc_UC, get_addr_confg  ; get address of the confg. reg.
   movb rl0, [r1]                 ; copy confg. reg. into rl0
   orb rl0, #04h                  ; sets the XTD bit (bit 2)
   movb [r1], rl0                 ; of the configuration register

   pop r1
   pop r0
   ret


;-----------------------------------------------------------------
; Function: clears the XTD (type) bit in the configuration
;           register of the message object.
; Inputs:   object number is passed in rl0. Its value must be
;           between 1 and 15.
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
set_type_std:

   push r0
   push r1

   calla cc_UC, get_addr_confg  ; get address of the confg. reg.
   movb rl0, [r1]                 ; copy confg. reg. into rl0
   andb rl0, #0F8h              ; clears the direction bit (bit 3)
   movb [r1], rl0               ; of the configuration register

   pop r1
   pop r0
   ret
```

```
;-----------------------------------------------------------------
; Function: returns the address of the upper arbitration
;           register.
; Input:    object number is passed in r10.
; Output:   address in r1.
; Destroys: r1.
;-----------------------------------------------------------------
get_addr_up_arb:

   calla cc_UC, get_addr_ctl
   add r1, #UP_ARBITR_OFF
   ret


;-----------------------------------------------------------------
; Function: returns the address of the lower arbitration
;           register.
; Input:    object number is passed in r10.
; Output:   address in r1.
; Destroys: r1.
;-----------------------------------------------------------------
get_addr_lo_arb:

   calla cc_UC, get_addr_ctl
   add r1, #LO_ARBITR_OFF
   ret


;-----------------------------------------------------------------
; Function: returns the address of the configuration register.
; Input:    object number is passed in r10.
; Output:   address in r1.
; Destroys: r1.
;-----------------------------------------------------------------
get_addr_confg:

   calla cc_UC, get_addr_ctl    ; get address of the control reg.
   add r1, #MSG_CONFG_OFF         ; compute address of the confg.
   ret                            ; register


;-----------------------------------------------------------------
; Function: returns the starting address of the data segment.
; Input:    object number is passed in r10.
; Output:   address in r1.
; Destroys: r1.
;-----------------------------------------------------------------
get_addr_data:

   calla cc_UC, get_addr_ctl    ; get address of the control reg.
   add r1, #DATA_OFF              ; compute address of the data
   ret                            ; segment.
```

```
;-----------------------------------------------------------------
; Function: returns the address of the message control register.
; Input:    object number is passed in r10.
; Output:   address in r1.
; Destroys: r1.
;-----------------------------------------------------------------
get_addr_ctl:

   cmpb r10, #1
   jmpr cc_NE, addr_ctl_2
   mov r1, #MSG_CTL_1
   ret
addr_ctl_2:
   cmpb r10, #2
   jmpr cc_NE, addr_ctl_3
   mov r1, #MSG_CTL_2
   ret
addr_ctl_3:
   cmpb r10, #3
   jmpr cc_NE, addr_ctl_4
   mov r1, #MSG_CTL_3
   ret
addr_ctl_4:
   cmpb r10, #4
   jmpr cc_NE, addr_ctl_5
   mov r1, #MSG_CTL_4
   ret
addr_ctl_5:
   cmpb r10, #5
   jmpr cc_NE, addr_ctl_6
   mov r1, #MSG_CTL_5
   ret
addr_ctl_6:
   cmpb r10, #6
   jmpr cc_NE, addr_ctl_7
   mov r1, #MSG_CTL_6
   ret
addr_ctl_7:
   cmpb r10, #7
   jmpr cc_NE, addr_ctl_8
   mov r1, #MSG_CTL_7
   ret
addr_ctl_8:
   cmpb r10, #8
   jmpr cc_NE, addr_ctl_9
   mov r1, #MSG_CTL_8
   ret
addr_ctl_9:
   cmpb r10, #9
   jmpr cc_NE, addr_ctl_10
   mov r1, #MSG_CTL_9
   ret
```

```
addr_ctl_10:
   cmpb rl0, #0Ah
   jmpr cc_NE, addr_ctl_11
   mov r1, #MSG_CTL_10
   ret
addr_ctl_11:
   cmpb rl0, #0Bh
   jmpr cc_NE, addr_ctl_12
   mov r1, #MSG_CTL_11
   ret
addr_ctl_12:
   cmpb rl0, #0Ch
   jmpr cc_NE, addr_ctl_13
   mov r1, #MSG_CTL_12
   ret
addr_ctl_13:
   cmpb rl0, #0Dh
   jmpr cc_NE, addr_ctl_14
   mov r1, #MSG_CTL_13
   ret
addr_ctl_14:
   cmpb rl0, #0Eh
   jmpr cc_NE, addr_ctl_15
   mov r1, #MSG_CTL_14
   ret
addr_ctl_15:
   mov r1, #MSG_CTL_15
   ret


;---------------------------------------------------------------
; Function: checks if a remote request has arrived. If so, sets
;           the remote request status flag, if not, clears it.
;           Upon the reception of the remote request it also
;           clears the RMTPND flag.
; Input:    object number in rl0.
; Output:   remote request status flag in rl7.
;           rl7 returns 1 if a remote request has been received.
;           rl7 returns 0 if a remote request has not been
;           received.
; Destroys: rl7.
;---------------------------------------------------------------
in_rmt_rqst:

   push r1
   push r2

   movb rl7, #0
   calla cc_UC, get_addr_ctl
   mov r2, [r1]
   and r2, #0F000h
   cmp r2, #0A000h
```

```
   jmpr cc_NE, no_rmt_rqst
   calla cc_UC, clr_rmtpnd
   calla cc_UC, clr_rxok
   movb rl7, #1

no_rmt_rqst:
   pop r2
   pop r1
   ret


;---------------------------------------------------------------
; Function: waits until a remote request arrives. Upon the
;           reception
;           of the remote request clears the RMTPND flag.
; Input:    object number in rL0.
; Output:   none.
; Destroys: none.
;---------------------------------------------------------------
wait_rmt_rqst:
   push r1
   push r2

   calla cc_UC, get_addr_ctl
wait1:
   mov r2, [r1]
   and r2, #0F000h
   cmp r2, #0A000h
   jmpr cc_NE, wait1
   calla cc_UC, clr_rmtpnd
   calla cc_UC, clr_rxok

   pop r2
   pop r1
   ret


;---------------------------------------------------------------
; Function: checks if a message has arrived. If so, sets the
;           message status flag, if not, clears it.
; Input:    object number in rl0.
; Output:   message status flag in rl7.
;           rl7 returns 1 if a message has been received.
;           rl7 returns 0 if a message has not been received.
; Destroys: rl7.
;---------------------------------------------------------------
in_msg:
   push r1

   movb rl7, #0
   calla cc_UC, get_addr_ctl
   mov r2, [r1]
```

```
   and r2, #0300h
   cmp r2, #0200h
   jmpr cc_NE, no_msg
   calla cc_UC, clr_rxok          ; clear RXOK
   movb rl7, #1

no_msg:
   pop r1
   ret


;---------------------------------------------------------------
; Function: waits until a message arrives.
; Input:    object number in rL0.
; Output:   none.
; Destroys: none.
;---------------------------------------------------------------
wait_msg:
   push r1

   calla cc_UC, get_addr_ctl
wait2:
   mov r2, [r1]
   and r2, #0300h
   cmp r2, #0200h
   jmpr cc_NE, wait2
   calla cc_UC, clr_rxok          ; clear RXOK

   pop r1
   ret


;---------------------------------------------------------------
; Function: sends a remote request.
; Input:    object number in rl0.
; Output:   none.
; Destroys: none.
;---------------------------------------------------------------
out_rmt_rqst:
   push r1

   calla cc_UC, set_txrq
   calla cc_UC, clr_cpuupd
rmt_rqst_not:
   mov r1, CTL_ST
   and r1, #0800h
   cmp r1, #0800h
   jmpr cc_NE, rmt_rqst_not
   calla cc_UC, clr_txok

   pop r1
   ret
```

```
;-----------------------------------------------------------------
; Function: sends a message.
; Input:    object number in R0.
; Output:   none.
; Destroys: none.
;-----------------------------------------------------------------
out_msg:

   push R1

   calla cc_UC, set_txrq
   calla cc_UC, clr_cpuupd
msg_not:
   mov R1, CTL_ST            ; see if txok is set,
   and R1, #0800h            ; i.e. if the message is succesfully
   cmp R1, #0800h            ; transmitted
   jmpr cc_NE, msg_not

   calla cc_UC, set_cpuupd     ; if so disable transmit (txrq is
   calla cc_UC, clr_txok       ; cleared by the CAN controller)
   calla cc_UC, clr_rxok

   pop R1
   ret
```

REFERENCES

[1] Altug, M. S., "Design and Performance Evaluation of Mixed MRP, Kanban and CONWIP production systems using Emulated Flexible Manufacturing Laboratory", *Master's Thesis*, University of Florida (1998).

[2] Anderson, J. A., and Rosenfeld, E., eds., "Neurocomputing: Foundations of Research", Cambridge: MIT Press (1988).

[3] Anderson, P. W., Arrow, K. J., and Pines, D., eds., "The Economy as an Evolving Complex System", *Santa Fe Institute Studies in the Sciences of Complexity*, Vol. 5 (1988).

[4] Arthur, B. W., et al., "Emergent Structures: A Newsletter of the Economic Research Program", *The Santa Fe Institute*, (March 1989 August 1990).

[5] Axelrod, R., "The Evolution of Cooperation", New York: Basic Books (1984).

[6] Badiru, A. B., "Expert Systems Applications in Engineering and Manufacturing", Englewood Cliffs, NJ: Prentice-Hall (1992).

[7] Balasubramanian S., Norrie, D. H., "A Multi-Agent Intelligent Design System Integrating Manufacturing and Shop-Floor Control", *ICMAS-95 Proceedings, First International Conference on Multi-Agent Systems*, pp. 3-9 (June 1995).

[8] Balch, T., Arkin, R. C., "Motor Schema-based Formation Control for Multi-Agent Robot Teams", *ICMAS-95 Proceedings, First International Conference on Multi-Agent Systems*, pp. 10-16 (June 1995).

[9] Brooks, R. "Elephants Don't Play Chess", *Robotics and Autonomous Systems* (1990).

[10] CAN Specification 2.0 Part B, Robert Bosch GmbH (Automotive Group).

[11] Cena G., "An Improved CAN Fieldbus for Industrial Applications", *IEEE Transactions on Industrial Electronics*, Vol. 44, no. 4, pp. 553-565 (August 1997).

[12] Chen, Q., and Luh, J. Y. S., "Coordination and Control of a Group of Small Mobile Robots", *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 2315-2320 (1994).

[13] Corbara, B., Drogoul, A., Fresnau, D., and Lalande, S. "Simulating the Sociogenesis process in Ant Colonies with MANTA", *Towards a Practice of Autonomous Systems II*, Cambridge: MIT Press (1993).

[14] Crowcroft, J., "Open Distributed Systems", Artech House Inc. (1995).

[15] Dallery, Y., and Gershwin, S. B., "Manufacturing Flow Line Systems: A Review of Models and Analytical results", *Queuing Systems*, Vol. 12, No. 3, pp. 3-93 (1992).

[16] Davies, P. C., ed., "The New Physics", New York: Cambridge University Press (1989).

[17] Day, J. D., and Zimmermann, H., "The OSI Reference Model", *Proceedings of the IEEE*, Vol. 71, pp. 1334-1340 (December 1983).

[18] Demezeau, Y., Boissier, O., and Koning, J. L., "Using Interaction Protocols to Control Vision Systems", *IEEE International Conference on Systems, Man and Cybernetics* (1994).

[19] Deneubourg, J. L., Aron, S., Goss, S., Pasteels, J. M., and Duerinck, G. "Random Behavior Amplification Processes and Number of Participants: How They Contribute To The Foraging Properties Of Ants", *Physica*, pp. 176-186 (1986).

[20] Deneubourg, J. L., Goss, S., Sendova-Franks, A., Detrain, C., and Chretien, L., "The Dynamics of Collective Sorting Robot-like Ants and Ant-like Robots", *From Animals to Animats*, pp. 356-363 (1991).

[21] Drogoul, A., "Can Strategies Emerge from Tactical Behaviors?", *5th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW-93*, pp. 13-27 (August 1993).

[22] ESPRIT Consortium, "CCE: An Integration Platform for Distributed Manufacturing Applications", Research Report, Project 7096, CCE-CNMA, Vol. 1 (1995).

[23] Fadiloglu, M., "A Matrix Polynomial Approach to Quasi-Birth-Death Process and its Implications on Models of Production Lines", PhD Thesis, Industrial and Systems Engineering, University of Florida, Gainesville, Florida (1998).

[24] Ferber, J., "Multi-Agent Systems", New York: Addison Wesley Longman (1999)

[25] Fischer, K., Mueller, J. P., Pischel, M., Schier, D., "A Model for Cooperative Transportation Scheduling", *ICMAS-95 Proceedings, First International Conference on Multi-Agent Systems*, pp. 109-116 (June 1995).

[26] Fleischmann, A., "Distributed Systems, Software Design & Implementation", Springer Verlag (1994).

[27] Forrest, S., "Emergent Computation: Self-Organizing, Collective, and Cooperative Phenomena in Natural and Artificial Computing Networks", Cambridge: MIT Press (1991).

[28] Ghedira, K., "Partial Constraint Satisfaction by a Multi-Agent Simulated Annealing Approach", *International Conference of AI, KBS, ES and NL*, Paris, (1994).

[29] Gleick , J., "Chaos: Making a New Science", New York: Penguin Books (1987).

[30] Gu, C., Ishida, T., "Analyzing the Social Behavior of Contract Net Protocol", 7^{th} *European Workshop on Modeling Autonomous Agents in a Multi-Agent World, MAAMAW-96*, pp. 116-127 (January 1996).

[31] Hodgson, T. J., and Wang. D., "Optimal Hybrid Push/Pull Control Strategies For A Parallel Multistage System Part I", *International Journal of Production Research*, Vol. 29, No. 6, pp. 1279-1287 (1991).

[32] Hogan, B. J., "Information shrinks the Factory", *Design News*, Vol. 51, No. 22, pp. 204-210 (Nov. 4 1996).

[33] Holland J. H., Holyoak, K. J., Hisbett, R. E., and Thagard. P. R., "Induction: Process of Inference, Learning and Discovery", Cambridge: MIT Press (1986).

[34] Hopp, W. J., and Spearman, M. L., "Factory Physics", Irwin Publishing, Chicago (1996).

[35] Huang, S. H., and Zhang, H. C., "Artificial Neural Networks in Manufacturing: Concepts , Applications and Perspectives", *IEEE Transactions on Components, Packaging, and Manufacturing Technology - Part* A, Vol. 17, No. 2, pp. 212-225 (June 1994).

[36] Huberman, B., Clearwater, S. H., "A Multi-Agent System for Controlling Building Environments", *ICMAS-95 Proceedings, First International Conference on Multi-Agent Systems*, pp.171-176 (June 1995).

[37] Iffenecker, C., and Ferber, J., "Using Multi-Agent Architecture for designing Electromechanical Products", *Proceedings of Avignon '92 Conference on Expert Systems and their Applications* (1992).

[38] Jennings, N., Corera, J. M., and Laresgoiti, I., "Developing Industrial Multi-Agent Systems", *First International Conference on Multi-Agent Systems*, MIT Press (1995).

[39] Kauffman, S. A., "Antichaos and Adaptation", *Scientific American*, pp. 78-84 (August 1991).

[40] Kauffman, S. A., "Origins of Order: Self-Organization and Selection in Evolution", Oxford: Oxford University Press (1992).

[41] Koeningsberg, E., "Production Lines and Internal Storage - A Review", *Management Science*, Vol. 5, pp. 410-433 (1959).

[42] Koopman, P. J., "Embedded System Design Issues (the Rest of the Story)", *Proceedings of the International Conference on Computer Design* (ICCD 96).

[43] Lanzola, G., Falasconi, S., and Stefanelli, M., "Cooperating Agents Implementing Distributed Patient Management", 7th *European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW-96*, pp. 218-232 (January 1996).

[44] Lesser, V. R., and Corkill, D. D., "The Distributed Vehicle Monitoring Testbed: A Tool for Investigating Distributed Problem Solving Networks", *AI Magazine*, Vol. 4, pp. 15-33 (1983).

[45] Lin, G. Y., and Solberg, J. J., "Integrated Shop Floor Control Using Autonomous Agents", *IEEE Transactions*, Vol. 24, No. 3, pp. 57-71 (July 1992).

[46] Liu, J. S., and Sycara K., "Emergent Constraint Satisfaction through Multi-Agent Coordinated Interaction", 5th *European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, MAAMAW-93 (1993).

[47] Luck, M., d'Inverno M., "A Formal Framework for Agency and Autonomy", *ICMAS-95 Proceedings, First International Conference on Multi-Agent Systems*, pp. 254-260 (June 1995).

[48] Madron, T. W., "LANS: Applications of IEEE / ANSI 802 Standards", New York: John Wiley & Sons Inc. (1989).

[49] Maio, D., and Rizzi, S., "Unsupervised Multi-Agent Exploration of Structured Environments", *ICMAS-95 Proceedings, First International Conference on Multi-Agent Systems*, pp. 269-275 (June 1995).

[50] Maira, A., "Local Area Networks - The Future of the Factory", *Manufacturing Engineering*, pp. 77-79 (March 1986).

[51] McGehee, J., Hebley, J., and Mahaffey, J., "The MMST Computer-Integrated Manufacturing System Framework", *IEEE Transactions on Semiconductor Manufacturing*, Vol. 7, No. 2, pp. 107-115 (May 1994).

[52] Mize, J. H., "Fundamentals of Integrated Manufacturing", *Guide to Systems Integration*, edited by J. Mize, Industrial Engineering and Management Press, Institute of Industrial Engineers, Norcross, Georgia, pp. 27-43 (1991).

[53] Mize, J. H., Bhuskute H. C., Pratt, D. B., and Kamath, M., "Modeling of Integrated Manufacturing Systems using an Object-Oriented Approach", *IIE Transactions*, Vol. 24, No. 3, pp. 14-25 (July 1992).

[54] Morley, R. E., and C. S., "An Analysis of a Plant-Specific Dynamic Scheduler", *Proceedings of the NSF Workshop on Dynamic Scheduling* (1993).

[55] Parsaei, H., and Jamshidi M., "Design and Implementation of Intelligent Manufacturing Systems", New Jersey: Prentice-Hall Inc. (1995).

[56] Perelson, A. S., and Kauffman S. A., eds., "Molecular Evolution on Rugged Landscapes: RNA, and the Immune System", *Santa Fe Institute Studies in the Sciences of Complexity*, Vol. 9 (1990).

[57] Pimentel J. R., "Communication Networks for Manufacturing", Engelwood Cliffs, NJ: Prentice-Hall Inc. (1990).

[58] Rosenthal, S., "Is C++ the not-ready-for-prime-time Embedded Language?", *Personal Engineering and Instrumentation News* (May 1992).

[59] Sabah, G., "CARAMEL: A Computational Model of Natural Language Understanding Using Parallel Implementation", *ECAI-90, Proceedings of the Ninth European Conference on Artificial Intelligence* (1990).

[60] Sadeh, N., "MICRO-BOSS: A micro-opportunistic factory scheduler", *Expert Systems Applications*, Vol. 6, No. 3, p. 377 (1993).

[61] Sargent, M., and Shoemaker, R. L., "The IBM PC from the Inside Out", Reading, MA: Addison Wesley (1984).

[62] Shaw, M., Solberg, J., and Woo, T. C., "System Integration in Intelligent Manufacturing: An Introduction", *IIE Transactions*, Vol. 24, No. 3, p. 2-5 (July 1992).

[63] Shaw, M. J., "A Distributed Scheduling Method for Computer Integrated Manufacturing: the Use of Local Area Networks in Cellular Systems", *International Journal of Production Research*, Vol. 25, No. 9, pp. 1285-1303 (1987).

[64] Sickle, T. V., "Programming Microcontrollers in C", Motorola's Series in Solid State Electronics.

[65] Siemens C167 Derivatives User's Manual 03.96 Version 2.0, Munchen: Siemens AG (1996).

[66] Sikora, R., and Shaw., M., "Coordination Mechanisms for Multi-Agent Manufacturing Systems: Applications to Integrated Manufacturing Scheduling", *IEEE Transactions on Engineering Management*, Vol. 44, No. 2, pp.175-187 (May 1997).

[67] Sipper, D., and Bulfin, R. L., "Production Planning, Control, and Integration", McGraw-Hill Companies, Inc. (1997).

[68] Smith, R. G., "The Contract net protocol: High-Level Communication and Control in a Distributed problem Solver", *IEEE Transactions on Computers*, C-29, Vol. 12, pp. 1104-1113 (December 1980).

[69] Smith, R. G., and Davis, R., "Frameworks for Cooperation in Distributed Problem Solving", *IEEE Transactions on Systems, Man and Cybernetics*, Vol. SMC-11, No. 1, pp. 61-70 (January 1981).

[70] Solberg J. J., "Production Planning and Scheduling in CIM", *Proceedings of the 11th World Computer Congress, IFIP* (August 1989).

[71] Spearman, M. L., Woodruff, D., and Hopp, W. J., "CONWIP: A Pull Alternative To Kanban", *International Journal of Production Research*, Vol. 18, No. 5, pp. 879-894 (1990).

[72] Steels, L., "Cooperation between Distributed Agents through Self-Organization", *Decentralized AI* (1989).

[73] Sycara, K., Roth, S., and Sadeh, N., "Resource allocation in distributed factory scheduling", *IEEE Expert*, Vol. 6, pp. 29-40 (Feb 1991).

[74] Tambe, M., "Recursive Agent and Agent-group Tracking in a Real-time Dynamic Environment", *ICMAS-95 Proceedings, First International Conference on Multi-Agent Systems,* pp.368-375 (June 1995).

[75] Tan, B., "A Decomposition Method for Multi-Station Production Systems", PhD Thesis, Industrial and Systems Engineering, University of Florida, Gainesville, Florida (1994).

[76] Tempelmeier, H., and Kuhn, H., "Flexible Manufacturing Systems", New York: John Wiley & Sons, Inc. (1993).

[77] Udo, G. J., "Neural Networks Applications in Manufacturing Processes", *Computers and Industrial Engineering,* Vol. 23, No. 1-4, pp. 97-100 (1992).

[78] Vail, P. S., "Computer Integrated Manufacturing", Boston: Pws-Kent Publishing Co. (1988).

[79] Valenti, M., "Machine tools get smarter", *Mechanical Engineering - CIME,* Vol. 117, No. 11, pp. 70-75 (November 1995).

[80] Waldrop, M. M., "Complexity: The Emerging Science at the Edge of Order and Chaos", New York: Simon & Schuster (1992).

[81] Walrand, J., "Communication Networks", Boston, MA: Aksen Associates Incorporated Publishers (1991).

[82] Wang, P., "Navigation Strategies for Multiple Autonomous Robots Moving in Formation", *Journal of Robotic Systems,* Vol. 8, pp. 177-195 (1991).

[83] Weihmayer, R. and Brandau, R., "A Distributed AI Architecture for Customer Network Control", *Globecom 90* (1990).

[84] Weiser, M., "The Computers in the 21st Century", *Scientific American,* Vol. 265, No. 3, pp. 94-104.

[85] Yeralan, S., and Ahluwalia, A., "Programming and Interfacing the 8051 Microcontroller", Reading, MA: Addison Wesley (1995).

[86] Zapfel, G., and Missbauer, H., "New Concepts For Production Planning and Control", *European Journal of Operational Research,* Vol. 67, No. 3, pp. 297-320 (1993).

[87] Zeghal, K., and Ferber, J., "CRAASH: A Coordinated Collision Avoidance System", *European Simulation Multiconference* (1993).

[88] Zuberi, K. M., Shin, K. G., "Real-time Decentralized Control with CAN", *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation*, pp. 93-99 (November 1996).

BIOGRAPHICAL SKETCH

Serdar Kırlı was born in İstanbul, Turkey, on April 5, 1967. He graduated from the Deutsche Schule in 1986. After receiving his B.S. degree in electrical engineering from Boğaziçi University, he attended the University of Florida for an M.E. degree in the same major. He continued his graduate studies in the Industrial and Systems Engineering Department where he received his Ph.D degree. His interests include industrial applications of microcontrollers, embedded control and object oriented programming.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Sencer Yeralan, Chair
Associate Professor of Industrial and Systems Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Süleyman Tüfekçi
Associate Professor of Industrial and Systems Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John F. Mahoney
Professor Emeritus of Industrial and Systems Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John K. Schueller
Associate Professor of Mechanical Engineering

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

August, 1999

M.J. Ohanian
Dean, College of Engineering

Winfred M. Phillips
Dean, Graduate School